Table of Contents

Porting/Building Code

Porting/Building Code: Overview

When you are developing code for NAS systems, testing a new compiler version, or porting a code from one platform to another, the following guidelines can help ensure that your code runs correctly and/or reproduces results from the platform it was ported from.

- Start with small problem sizes and use just a few time steps/iterations to quickly check whether the program is running correctly; setting up your PBS script and data files can often be done with 10 minute jobs.
- Run your test jobs on the PBS **`debug`** queue for quicker turnaround.
- Make as few code changes as possible when porting, and use the same data sets on both old and new platforms to compare results.
- Keep in mind that an absence of error messages does not necessarily mean that the code is running correctly on either platform.
- Be attentive to porting data files: for example, unformatted Fortran files (**`FORM='unformatted')`** are not necessarily portable.
- If there are any problems with code ported to a new platform, do not assume that the new code is wrong and the previous code is rightâ   both may be wrong.

The articles in the Porting/Building Code section provide information about compilers and libraries, as well as methods for porting, debugging, and analyzing performance.

Compiling

Endian and Related Environment Variables or Compiler Options

Intel Fortran expects numeric dataâ   both integer and floating-point dataâ   to be in native little-endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

If your program needs to read or write unformatted data files that are not in little-endian order, you can use one of the following six methods, which are provided by Intel. The methods are listed in the order of precedence.

Method 1. Setting a Variable for a Specific Unit Number

Set an environment variable for a specific unit number before the file is opened. The environment variable is named **FORT_CONVERT*n***, where *n* is the unit number. For example:

```
setenv FORT_CONVERT28 BIG_ENDIAN
```

No source code modification or recompilation is needed.

Method 2. Setting A Variable for a Specific File Name

Set an environment variable for a specific file name extension before the file is opened. The environment variable is named **FORT_CONVERT.ext** or **FORT_CONVERT_ext**, where "ext" is the file name extension (suffix). The following example specifies that a file with an extension of ".dat" is in big-endian format:

```
setenv FORT_CONVERT.DAT BIG_ENDIAN
```

Some Linux command shells may not accept a dot (.) for environment variable names. In that case, use **FORT_CONVERT_ext** instead.

No source code modification or recompilation is needed.

Method 3. Setting a Variable for a Set of Units

Set an environment variable for a set of units before any files are opened. The environment variable is named **F_UFMTENDIAN**.

Syntax

```
Csh: setenv F_UFMTENDIAN MODE;EXCEPTION

Sh : export F_UFMTENDIAN=MODE;EXCEPTION

MODE = big | little

EXCEPTION = big:ULIST | little:ULIST | ULIST

ULIST = U | ULIST,U

U = decimal | decimal-decimal
```

MODE defines the current format of the data, represented in the files; it can be omitted. The keyword "little" means that the data has little-endian format and will not be converted. For IA-32 systems, this keyword is a default. The keyword "big" means that the data has big-endian format and will be converted. This keyword may be omitted together with the colon.

EXCEPTION is intended to define the list of exclusions for MODE; it can be omitted. EXCEPTION keyword ("little" or "big") defines data format in the files that are connected to the units from the EXCEPTION list. This value overrides MODE value for the units listed.

Each list member U is a simple unit number or a number of units. The number of list members is limited to 64. decimal is a non-negative decimal number less than 2**32.

The environment variable value should be enclosed in quotes if the semicolon is present.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Examples

```
setenv F_UFMTENDIAN big
```

All input/output operations perform conversion from big-endian to little-endian order on READ, and from little-endian to big-endian order on WRITE.

```
setenv F_UFMTENDIAN "little;big:10,20"
```

```
or setenv F_UFMTENDIAN big:10,20
```

```
or setenv F_UFMTENDIAN 10,20
```

In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

```
setenv F_UFMTENDIAN "big;little:8"
```

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

```
setenv F_UFMTENDIAN 10-20
```

Define 10, 11, 12, ...19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

Method 4. Using the CONVERT Keyword in the OPEN Statement

Specify the **`CONVERT`** keyword in the OPEN statement for a specific unit number. Note that a hard-coded **`OPEN`** statement **`CONVERT`** keyword value cannot be changed after compile time. The following **`OPEN`** statement specifies that the file **`graph3.dat`** is in **`VAXD`** unformatted format:

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
```

Method 5. Compiling with an OPTIONS Statement

Compile the program with an **`OPTIONS`** statement that specifies the **`CONVERT=`*keyword*** qualifier. This method affects all unit numbers using unformatted data specified by the program. For example, to use **`VAX F_floating`** and **`G_floating`** as the unformatted file format, specify the following **`OPTIONS`** statement:

```
OPTIONS /CONVERT=VAXG
```

Method 6. Compiling with the -convert keyword Option

Compile the program with the command-line **`-convert keyword`** option, which affects all unit numbers that use unformatted data specified by the program. For example, the following command line compiles program **`file.for`** to use **`VAXD`** floating-point data for all unit numbers:

```
ifort file.for -o vconvert.exe -convert vaxd
```

In addition, if the record length of your unformatted data is in byte units (Intel Fortran default is in word units), use the **`-assume byterecl`** compiler option when compiling your source code.

GNU Compiler Collection

The GNU Compiler Collection (GCC) is an integrated distribution of compilers for several major programming languages, which currently include C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada.

The GNU C and C++ compiler (**`gcc and g++`**) and Fortran compiler (**`gfortran`**) are available on NAS systems through the Linux operating system distribution. The current version, which is installed in the **`/usr/bin`** directory, can be found by using the **`gcc -v`** command. For example:

```
% gcc -v
[...]
4.8.5 20150623 (Red Hat 4.8.5-44) (GCC)
```

To see which GCC versions are available as modules on NAS systems, run **`module avail`**.

Read **man gcc** and **man gfortran** for more information.

OpenMP

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for various platforms.

Intel version 11.*x* compilers support OpenMP spec-3.0 while 10.*x* compilers support spec-2.5.

Building OpenMP Applications

The following Intel compiler options can be used for building or analyzing OpenMP applications:

- **`-openmp`**

  Enables the parallelizer to generate multithreaded code based on OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. The **`-openmp`** option works with both **`-O0`** (no optimization) and any optimization level of **`-O`**. Specifying **`-O0`** with **`-openmp`** helps to debug OpenMP applications.

  Note that setting **`-openmp`** also sets **`-automatic`**, which causes all local, non-SAVEd variables to be allocated to the run-time stack, which may provide a performance gain for your applications. However, if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. If you want to cause variables to be placed in static memory, specify option **`-save`**. If you want only scalar variables of certain intrinsic types (integer, real, complex, logical) to be placed on the run-time stack, specify option **`-auto-scalar`**.
- **`-assume cc_omp`** or **`-assume nocc_omp`**

  `-assume cc_omp` enables conditional compilation as defined by the OpenMP Fortran API. That is, when "!$space" appears in free-form source or "c$spaces" appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.

  `-assume nocc_omp` tells the compiler that conditional compilation as defined by the OpenMP Fortran API is disabled unless option **`-openmp`** (Linux) or **`/Qopenmp`** (Windows) is specified.
- **`-openmp-lib legacy`** or **`-openmp-lib compat`**

  Choosing **`-openmp-lib legacy`** tells the compiler to use the legacy OpenMP run-time library (libguide). This setting does not provide compatibility with object files created using other compilers. This is the default for Intel version 10.*x* compilers.

  Choosing **`-openmp-lib compat`** tells the compiler to use the compatibility OpenMP run-time library (libiomp). This is the default for Intel version 11.*x* compilers.

  On Linux systems, the compatibility Intel OpenMP run-time library lets you combine OpenMP object files compiled with the GNUgcc or gfortran compilers with similar OpenMP object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

  You cannot link object files generated by the Intel Fortran compiler to object files compiled by the GNU Fortran compiler, regardless of the presence or absence of the **`-openmp`** (Linux) or /Qopenmp (Windows) compiler option. This is because the Fortran run-time libraries are incompatible.

  Note: The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compiler earlier than 10.0.

- **`-openmp-link dynamic`** or **`-openmp-link static`**

  Choosing **`-openmp-link dynamic`** tells the compiler to link to dynamic OpenMP run-time libraries. This is the default for Intel version 11.*x* compilers.

  Choosing **`-openmp-link static`** tells the compiler to link to static OpenMP run-time libraries.

  Note that the compiler options **`-static-intel`** and **`-shared-intel`** have no effect on which OpenMP run-time library is linked.

  Note that this option is only available for newer Intel compilers (version 11.*x*).
- **`-openmp-profile`**

  Enables analysis of OpenMP applications. To use this option, you must have Intel(R) Thread Profiler installed, which is one of the Intel(R) Threading Tools. If this threading tool is not installed, this option has no effect.

  Note that Intel Thread Profiler is not installed on Pleiades.
- **`-openmp-report[`*`n`*`]`**

  Controls the level of diagnostic messages of the OpenMP parallelizer. *n*=0,1,or 2.
- **`-openmp-stub`**

  Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

OpenMP Environment Variables

There are a few OpenMP environment variables one can set. The most commonly used are:

- **OMP_NUM_THREADS *num***

  Sets number of threads for parallel regions. Default is 1 on Pleiades. Note that you can use **`ompthreads`** in the PBS resource request to set values for OMP_NUM_THREADS. For example:

  ```
  %qsub -I -lselect=1:ncpus=4:ompthreads=4
  Job 991014.pbspl1.nas.nasa.gov started on Sun Sep 12 11:33:06 PDT 2010
  ...
  PBS r3i2n9> echo $OMP_NUM_THREADS
  4
  PBS r3i2n9>
  ```
- **OMP_SCHEDULE *type*[,*chunk*]**

  Sets the run-time schedule type and chunk size. Valid OpenMP schedule types are static, dynamic, guided, or auto. Chunk is a positive integer.
- **OMP_DYNAMIC *true*** or **OMP_DYNAMIC *false***

  Enables or disables dynamic adjustment of threads to use for parallel regions.
- **OMP_STACKSIZE *size***

  Specifies size of stack for threads created by the OpenMP implementation. Valid values for size (a positive integer) are *size, size*B, *size*K, *size*M, *size*G. If units B, K, M or G are not specified, size is measured in kilobytes (K).

  Note that this feature is included in OpenMP spec-3.0, but not in spec-2.5.

Note that Intel also provides a few additional environment variables. The most commonly used are:

- **KMP_AFFINITY *type***

  Binds OpenMP threads to physical processors. Available *type*: compact, disabled, explicit, none, and scatter.

  WARNING: There is a conflict between KMP_AFFINITY in Intel 11.*x* runtime library and **`dplace`**, causing all threads to be placed on a single CPU when both are used. It is recommended that KMP_AFFINITY be set to *disabled* when using **`dplace`**.
- **KMP_MONITOR_STACKSIZE**

  Sets stack size in bytes for monitor thread.
- **KMP_STACKSIZE**

  Sets stack size in bytes for each thread.

For more information, please see the official OpenMP web site.

PGI Compilers and Tools

As an alternative to using Intel compilers for Fortran, C, and C++ programs, you can use the compilers and program development tools from PGI Compilers and Tools. By default, the PGI compilers generate code that is optimized for the compilation host.

PGI Compiling

Several versions of PGI compilers and tools are installed as modules under the **/nasa** system directory on Pleiades and Endeavour. To view available versions, run:

```
pfe21% module use -a /nasa/modulefiles/testing
pfe21% module avail comp-pgi
--------- /nasa/modulefiles/sles12 ---------
comp-pgi/16.10 comp-pgi/17.1

-------- /nasa/modulefiles/testing ---------
comp-pgi/17.10 comp-pgi/18.10 comp-pgi/18.4  comp-pgi/19.10 comp-pgi/19.5  comp-pgi/20.4
```

The following command line will load version 18.10:

```
pfe21% module load comp-pgi/18.10
```

Note: Starting with the PGI compiler 2018, two different code generators (PGI-proprietary vs LLVM-based) are included in each package. The default code generator for 2018 versions is the PGI-proprietary code generator and for 2019 and later versions is the LLVM-based generator. The PGI 2019 compilers using LLVM are not link-compatible with PGI 2018 and prior releases.

Using the PGI Compilers

PGI provides multiple commands for different languages or functions. In 18.10 and earlier versions, the following are available.

Command	Language or Function
pgfortran	PGI Fortran
pgf77	Fortran 77
pgf90 or pgf95	Fortran 90/95/F2003
pgcc	ANSI C99 and K&R (Kernighan and Ritchie) C
pgc++	GNU-compatible C++
pgdbg	Source code debugger (supports OpenMP and MPI)
pgprof	Performance profiler (supports OpenMP and MPI)

Note: By default, the PGI compilers generate code that is optimized for the type of processor on which compilation is performed (the compilation host). Be aware that the processors are forward-compatible, but not backward-compatible. Thus a code compiled and optimized on a newer-generation processor will not necessarily execute correctly on previous-generation processors.

If you want to build an executable that targets a specific processor type, use the **-tp** flag:

-tp= skylake-64	

	Intel Skylake Xeon processor, 64-bit mode (includes Skylake and Cascade Lake)
-tp= haswell-64	Intel Haswell architecture Core processor, 64-bit mode (includes Haswell and Broadwell)
-tp= sandybridge-64	Intel SandyBridge architecture Core processor, 64-bit mode (includes Sandy Bridge and Ivy Bridge)

TIP: Using the **`-tp=sandybridge-64,haswell-64,skylake-64`** option will generate a single executable in which the code is optimized for the Intel Sandy Bridge, Haswell, and Skylake architectures. The choice of which optimized copy to execute is made at run time and depends on the machine executing the code. You must load **`comp-pgi/18.4`** or later versions in order to use **`-tp= skylake-64`**. PGI recommends that for best performance on processors which support SSE instructions (including all NAS processor types), use **`pgfortran`**, even for FORTRAN 77 code, use the **`-fastsse`** option.

For more information about the PGI compilers, see **man pgfortran, man pgcc,** and **man pgc++** or use the **`pgfortran -help`**, **`pgcc -help`**, or **`pgc++ -help`** commands. Information about the PGI debugger and performance analysis tool can be found using the commands **`pgdbg -help`** and **`pgprof -help`**.

Intel Compiler

Intel compilers are recommended for building your applications on HECC systems. There is no default compilerâ   you must load a specific module. To see what modules are available, run the **`module avail`** command:

```
pfe21% module avail
```

In the output, look for the **`comp-intel/`*`versions`*** entries. Then, load the Intel compiler module you want to use. For example:

```
pfe21% module load comp-intel/2018.3.222
```

When you load a compiler module, some environment variables (such as PATH, CPATH, and LD_LIBRARY_PATH) are set or modified to add the paths to certain commands, include files, or libraries to your environment. This helps to simplify the way you do your work.

You can use the **`module show`** command check what environment variables will be modified for a particular module. For example:

```
pfe21% module show comp-intel/2018.3.222
```

Intel Compilers for HECC Systems

On HECC systems, there are Intel compilers for both Fortran and C/C++:

Intel Fortran Compiler: ifort (version 8 and above)

The **`ifort`** command invokes the Intel Fortran Compiler to preprocess, compile, assemble, and link Fortran programs.

```
pfe21% ifort [options] file1 [file2 ...]
```

Read **man ifort** for all available compiler options. To see the compiler options by categories, run:

```
pfe21% ifort -help
```

file*N* is a Fortran source (**`.f .for .ftn .f90 .fpp .F .FOR .FTN .FPP .F90 .i .i90`**), assembly (**`.s .S`**), object (**`.o`**), static library (**`.a`**), or other linkable file. Interpret the suffixes of the Fortran source files as follows:

- **`.f, .for, or .ftn`** = Fixed-form source files.
- **`.f90`** = Free-form F95/F90 source files.
- **`.fpp, .F, .FOR, .FTN, or .FPP`** = Fixed-form source files that must be pre-processed by the FPP pre-processor before being compiled.
- **`.F90`**: = Free-form Fortran source files that must be pre-processed by the FPP pre-processor before being compiled.

Intel C/C++ Compiler: icc and icpc Commands (version 8 and above)

The Intel C++ Compiler is designed to process C and C++ programs on Intel-architecture-based systems. You can preprocess, compile, assemble, and link these programs.

```
pfe21% icc [options] file1 [file2 ...]
% icpc [options] file1 [file2 ...]
```

Read **man icc** for all available compiler options. To see the compiler options by categories, run:

```
pfe21% icc -help
```

The **`icpc`** command uses the same compiler options as the **`icc`** command, with the following differences:

- Invoking the compiler using **`icpc`** compiles .c and .i files as C++, and always links in C++ libraries.
- Invoking the compiler using **`icc`** compiles .c and .i files as C, and only links in C++ libraries if C++ source is provided on the command line.

file*N* represents a C/C++ source (**`.C .c .cc .cp .cpp .cxx .c++ .i`**), assembly (**`.s`**), object (**`.o`**), static library (**`.a`**), or other linkable file.

Recommended Compiler Options

Intel compilers are recommended for building your applications on HECC systems. There is no default compilerâ you must load a specific module. To see what modules are available, run the **`module avail`** command:

```
pfe21% module avail
```

To load a specific version, replace *version* in the command below with the actual version number:

```
pfe21% module load comp-intel/version
```

If you have no preference of which version to use, you can start with the latest version:

```
pfe21% module load comp-intel/2020.4.304
```

Some important compiler flags to keep in mind are described in the following sections.

Turn On Optimization

If you do not specify an optimization level (**`-On, n=0,1,2,3`**), the default is **`-O2`**. If you want more aggressive optimizations, you can use **`-O3`**.

Note: Using **`-O3`** may not improve performance for some programs.

Generate Optimized Code for a Processor Type

Intel compilers provide flags for generating optimized codes for various instruction sets used in specific processors or micro-architectures. If your goal is to achieve the best performance from a specific processor, we recommend that you build your application using the latest Intel compiler with the flag that corresponds with the processor you plan to use:

Processor Type	System	Flag
Cascade Lake	Aitken/Endeavour	**`-xCORE-AVX512`**
Skylake	Electra	
Broadwell	Electra/Pleiades	**`-xCORE-AVX2`**
Haswell	Pleiades	
Ivy Bridge	Pleiades	**`-xAVX`**
Sandy Bridge	Pleiades	

Note: The instruction sets are upward compatible, therefore, applications compiled with:

- **`-xAVX`** can run on Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Cascade Lake processors.
- **`-xCORE-AVX2`** can run on Haswell, Broadwell, Skylake, and Cascade Lake processors.
- **`-xCORE-AVX512`** can run *only* on Skylake and Cascade Lake processors.

WARNING: Executables built with a flag that is not compatible with a processor will not run on that processor, and will result in an error with a message similar to the following:

```
Fatal Error: This program was not built to run in your system. Please verify that both the operating syst
```

AMD Rome processors: See Preparing to Run on Aitken Rome Nodes for information about compilers to use for Rome processors.

Generate Code for Any Processor Type

If your goal is to create a portable executable file that can run on any processor type on Pleiades, Electra, Aitken, and Endeavour, you can choose one of the following approaches:

- Use none of the above flags (which defaults to **`-mSSE2`**)
- Use **`-O3 -axCORE-AVX512,CORE-AVX2 -xAVX`** (with the latest Intel compiler)

This allows a single executable that will run on any of the processor types, with suitable optimization to be determined at run time.

WARNING: Be careful if you plan to add the **`-Ofast`** flag, which sets **`-xHost`** (and other options) automatically during compilation. If you place **`Ofast`** after **`-axCORE-AVX512,CORE-AVX2 -xAVX`**, **`-xHost`** takes precedence over **`-axCORE-AVX512,CORE-AVX2 -xAVX`**, and the executable is generated for the compilation host processor.

Turn On Inlining

Use the **`-ip`** or **`-ipo`** flags.

Using **`-ip`** enables additional interprocedural (IP) optimizations for single-file compilation. One of these optimizations enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

Using **`-ipo`** enables multi-file IP optimizations between files. When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

Use a Specific Memory Model

If you get a link time error relating to R_X86_64_PC32, add the compiler option **`-mcmodel=medium`** and the link option **`-shared-intel`**. This happens if a common block is more than 2 GB in size.

Turn Off All Warning Messages

Using the **`-w`** option disables all warnings; **`-vec-report0`** disables printing of vectorizer diagnostic information; and **`-opt-report0`** disables printing of optimization reports.

Parallelize Your Code

The **`-qopenmp`** option handles OMP directives and **`-parallel`** looks for loops to parallelize.

For more compiler/linker options, see **man ifort, man icc**.

Debugging Options

See Recommended Intel Compiler Debugging Options for descriptions of several more compiler flags.

Porting to NAS Systems

Porting with HPE MPT

Among the Message Passing Interface (MPI) libraries installed on Pleiades, we recommend using HPE's Message Passing Toolkit (MPT) library. Use the command **`module avail mpi-hpe`** to determine which MPT versions are available. You can always load the default MPT version by running **`module load mpi-hpe/mpt`**.

Be sure to load the module in both of the following places:

- On the front-end node when you build your application.
- In the PBS script you use to run jobs on the compute nodes.

Network Considerations

NAS systems use an InfiniBand (IB) network for inter-process remote direct memory access (RDMA) communications. There are two InfiniBand fabrics, designated as ib0 and ib1. In order to maximize performance, we recommend using the ib0 fabric for all MPI traffic. The ib1 fabric is reserved for storage-related traffic. The default configuration for MPT on NAS systems is to use only the ib0 fabric, although the MPI_IB_RAILS environment variable is set to 1+ to allow failover to the ib1 fabric if there are errors using ib0.

TIP: Loading **`mpi-hpe/mpt`** instead of an explicit MPT modulefile allows you to always get the NAS-recommended MPT version and environmental variable settings that enable improvements in InfiniBand stability and scalability.

Environment Variables

When you load an MPT module, several paths (such as CPATH and LD_LIBRARY_PATH) and MPT-related environment variables are reset or modified to non-default values. You can also set other environment variables that may be useful for some applications or for debugging purposes.

Modified Environment Variables

The variables that are modified when you load a module might vary by module version. To find out what environmental variables are set by the modulefile, use the **`module show`** command as follows:

```
% module show mpt_modulefile_name
```

where *mpt_modulefile_name* refers to an explicit MPT modulefile that **`mpi-hpe/mpt`** points to. Some of the variables that are likely to be set to a non-HPE-default value include:

```
MPI_IB_TIMEOUT
MPI_IB_RAILS
MPI_IB_FAILOVER
MPI_IB_RECV_MSGS
MPI_IB_RNR_TIMER
MPI_SYSLOG_COPY
MPI_UD_RECV_MSGS
MPI_UD_TIMEOUT
MPI_WATCHDOG_TIMER
```

`MPIO_LUSTRE_GCYC_MIN_ITER`

The descriptions of these variables and their HPE default values are as follows:

MPI_IB_TIMEOUT
: When an IB card sends a packet, it waits some amount of time for an acknowledgement (ACK) packet to be returned by the receiving IB card. If it does not receive one, it sends the packet again. This variable controls the wait period. The time spent is equal to 4 * 2 ^ MPI_IB_TIMEOUT microseconds.
(HPE) Default: 18

MPI_IB_RAILS
: If the MPI library uses the IB driver as the inter-host interconnect, it will by default use a single IB fabric. If this variable is set to 2, the library will try to make use of multiple available separate IB fabrics (ib0 and ib1) and split its traffic across them. If the fabrics do not have unique subnet IDs, then the rail-config utility is required to have been run by the system administrator to enable the library to correctly use the separate fabrics. If this variable is set to 1+, then MPT will set up IB connection on two rails but only send traffic on one; the second rail will be reserved for failover in case of errors.
(HPE) Default: 1

MPI_IB_FAILOVER
: When the MPI library uses IB and a connection error is detected, the library will handle the error and restart the connection a number of times equal to the value of this variable. Once there are no more failover attempts left and a connection error occurs, the application will be aborted.
(HPE) Default: 0

MPI_IB_RECV_MSGS
: When the MPI library uses IB it must allocate some number of buffers to receive data headers and short messages into. If a rank has all its buffers filled before it can service them, it may go into an error mode. If this happens, try doubling the value of this variable.
(HPE) Default: 512

MPI_IB_RNR_TIMER
: When a packet arrives at an InfiniBand HCA and there are no remaining receive buffers for it, the receiving HCA sends a NAK to the requestor. The requesting HCA retries again after some period of time. This variable controls the delay time. Setting a higher value may slow performance in some circumstances, but it will also likely significantly help fabric health during high congestion. See Table 45 of the official InfiniBand specification for precise translations of this value to delay times.
(HPE) Default: 14

MPI_SYSLOG_COPY
: When this variable is set, messages about MPT internal errors and system failures are copied to the system log on the machines where they happened. If this variable is set to "2" then MPT internal warnings will also be copied to the syslog.
(HPE) Default: Not set

MPI_UD_RECV_MSGS
: This controls the number of InfiniBand UD receive buffers.
(HPE) Default: 800

MPI_UD_TIMEOUT
: The IB UD code uses this variable as a timeout to control when to resend UD packets if it has not received some form of ACK from the receiver. The variable is in units of milliseconds.
(HPE) Default: 20

MPI_WATCHDOG_TIMER
: If an MPT process has been having continuous IB fabric problems and has not been able to contact another process for this many minutes, then it will abort the application. By default MPT keeps trying to make transfers even in the face of flaky fabrics. This variable

may help abort the job sooner if a node has crashed. Set to "0" to disable.
(HPE) Default: 10

MPIO_LUSTRE_GCYC_MIN_ITER
Enables group-cyclic aggregator assignment. Group-cyclic allows multiple aggregators to write to each OST while minimizing lock contention. This value specifies the minimum number of stripes that each aggregator will write so as to ensure the added lock contention of more aggregators is offset by the added bandwidth of more data written per two-phase iteration. A value of "1" will allow assignment of the maximum possible number of aggregators.
(HPE) Default: 0 (not enabled)

See **man MPI** for more information.

Additional Environment Variables

For more MPT-related variables, read **man MPI** after you load an MPT module. Some variables may be useful for certain applications or for debugging purposes. Here are a few of them for you to consider:

MPI_DSM_DISTRIBUTE
Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. This feature can also be overridden by using `dplace` or `omplace`. This feature is most useful if running on a dedicated system or running within a cpuset.
Default: Enabled

MPI_MEMMAP_OFF
Turns off the memory mapping feature, which provides support for single-copy transfers and MPI-2 one-sided communication on Linux. These features are supported for single-host MPI jobs and MPI jobs that span partitions. At job startup, MPI uses the `xpmem` module to map memory from one MPI process onto another. The mapped areas include the static region, private heap, and stack. Memory mapping is enabled by default on Linux. To disable it, set the MPI_MEMMAP_OFF environment variable.
Default: Not enabled

MPI_BUFS_PER_PROC
Determines the number of private message buffers (16 KB each) that MPI is to allocate for each process. These buffers are used to send and receive long messages. Each process has this many buffers for sending with and this many for receiving with. The memory consumed on a shepherd is #ranks_in_the_shepherd * MPI_BUS_PER_PROC * 2 * 16k.
Default: 128 buffers (1 page = 16KB)

MPI_COREDUMP
Controls which ranks of an MPI job can dump core on receipt of a core-dumping signal. Valid values are NONE, FIRST, ALL, or INHIBIT.
NONE means that no rank should dump core.
FIRST means that the first rank on each host to receive a core-dumping signal should dump core.
ALL means that all ranks should dump core if they receive a core-dumping signal.
INHIBIT disables MPI signal-handler registration for core-dumping signals.
Default: FIRST

MPI_STATS (toggle)
Enables printing of MPI internal statistics. Each MPI process prints statistics about the amount of data sent with MPI calls during the MPI_Finalize process.
Default: Not enabled

MPI_DISPLAY_SETTINGS

If set, MPT will display the default and current settings of the environment variables controlling it.
Default: Not enabled

MPI_VERBOSE
: Setting this variable causes MPT to display information such as what interconnect devices are being used and what environment variables have been set by the user to non-default values. Setting this variable is equivalent to passing **mpirun** the **-v** option.
Default: Not enabled

MPI_IB_XRC
: When MPT is creating IB connections lazily and using IB host channel adapters (HCAs) that support extended reliable connection (XRC) queue pairs (QPs), MPT will use XRC QPs instead of regular RC QPs. XRC QPs are a more scalable version of the RC QPs currently found on ConnectX IB HCAs.
Default: Enabled

MPI_IB_NUM_QPS
: When InfiniBand RC QPs are being allocated lazily, this limits the number of QPs that any one rank may use. This helps to prevent the host from completely running out of QPs.
Default: (Maximum number of QPs the HCAs can support - 2k) / # of local ranks

MPI_IB_QTIME
: MPT has the ability to use the controlled delay (CoDel) traffic throttling mechanism with Reliable Connection (RC) and User Datagram (UD) InfiniBand protocols. Cluster nodes have the ability to saturate InfiniBand fabrics to the state that they become backed up, start dropping packets, and break all sorts of services like TCP/IP control channels and Lustre. If enabled, MPT will monitor the approximate time that outgoing transfers are queued up at the HCA. If queue time exceeds the value of this variable, then MPT will take measures to significantly reduce the amount of traffic that it is putting onto the fabric for a short period of time. In many cases, this will eventually reduce fabric loading to a reasonable level. This variable is in units of microseconds (us); for example, a value of "200" will cause throttling to begin if minimum queue time exceeds 200us. Suggested values to try are in the range of 100us - 5000us.
Default: Disabled

Building Applications

To build MPI applications with the HPE MPT library, link with **-lmpi** and/or **-lmpi++**. See HPE MPT for some examples.

Running Applications

MPI executable files built with HPE MPT are not allowed to run on the Pleiades front-end (PFE) nodes.

You can run your MPI job on the compute nodes in an interactive PBS session or through a PBS batch job. After loading an MPT module, use the **mpiexec** command to start your MPI processes (do not use **mpirun**).

For example:

```
#PBS -lselect=2:ncpus=20:mpiprocs=20:model=ivy
....
module load mpi-hpe/mpt
mpiexec -np N ./your_executable
```

The **-np** flag (with *N* MPI processes) can be omitted if the value of *N* is the same as the product of the value specified for **select** and the value specified for **mpiprocs**.

Performance Considerations

If your MPI job uses all of the cores in each node (16 MPI processes/node for Sandy Bridge, 20 MPI processes/node for Ivy Bridge, 24 MPI processes/node for Haswell, 28 processes/node for Broadwell, 40 processes/node for Skylake and Cascade Lake, and 128 processes/node for Rome), then pinning MPI processes greatly helps the performance of the code.

By default, the environment variable MPI_DSM_DISTRIBUTE is set to 1 (or true), which will pin processes consecutively on the cores.

If your MPI job does *not* use all the cores in each node, we recommend that you disable MPI_DSM_DISTRIBUTE as follows:

```
setenv MPI_DSM_DISTRIBUTE 0
```

Then, let the Linux kernel decide where to place your MPI processes. If you want to pin processes explicitly, you can try using one of the methods described in Process/Thread Pinning Overview.

The following recommended process pinning method uses the mbind.x tool to pin an eight-MPI-process job with every four processes on four processor cores of a node, using two nodes:

```
#PBS -lselect=2:ncpus=4:mpiprocs=4:model=xxx
mpiexec -np 8 mbind.x ./your_executable
```

Porting with HPE's MPI and Intel OpenMP

Building Applications

To build an MPI/OpenMP hybrid executable using HPE's MPT and Intel's OpenMP libraries, you must compile your code with the **-openmp** flag and link it with the **-lmpi** flag. For example:

```
%module load comp-intel/2020.4.304 mpi-hpe/mpt
%ifort -o your_executable prog.f -openmp -lmpi
```

Running Applications

Here is a sample PBS script for running an MPI/OpenMP application on Pleiades using 3 nodes, with 10 MPI processes on each node and 2 OpenMP threads per MPI process.

```
#PBS -lselect=3:ncpus=20:mpiprocs=10:model=ivy
#PBS -lwalltime=1:00:00

module load comp-intel/2020.4.304 mpi-hpe/mpt
setenv OMP_NUM_THREADS 2

cd $PBS_O_WORKDIR

mpiexec ./your_executable
```

You can specify the number of threads on the PBS resource request line using **ompthreads**. This causes the PBS prologue to set the **OMP_NUM_THREADS** environment variable.

```
#PBS -lselect=3:ncpus=20:mpiprocs=10:ompthreads=2:model=ivy
#PBS -lwalltime=1:00:00

module load comp-intel/2020.4.304 mpi-hpe/mpt

cd $PBS_O_WORKDIR

mpiexec ./your_executable
```

Performance Issues

For pure MPI codes built with HPE's MPT library, you can improve performance on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell nodes by pinning the processes through setting **MPI_DSM_DISTRIBUTE** environment variables to 1 (or true). However, for MPI/OpenMP codes, all the OpenMP threads for the same MPI process have the same process ID. In this case, setting this variable to 1 causes all OpenMP threads to be pinned on the same core and the performance suffers.

It is recommended that you set **MPI_DSM_DISTRIBUTE** to 0 and use **omplace** for pinning instead.

If you use Intel version 10.1.015 or later, you should also set **KMP_AFFINITY** to *disabled* or **OMPLACE_AFFINITY_COMPAT** to ON as Intel's thread affinity interface would interfere with **dplace** and **omplace.**

```
#PBS -lselect=3:ncpus=20:mpiprocs=10:ompthreads=2:model=ivy
#PBS -lwalltime=1:00:00

module load comp-intel/2020.4.304 mpi-hpe/mpt

setenv MPI_DSM_DISTRIBUTE 0
setenv KMP_AFFINITY disabled
```

```
cd $PBS_O_WORKDIR

mpiexec -np 30 omplace ./your_executable
```

Porting with Intel-MPI

You can use the Intel MPI library to build and run your MPI application. The Intel MPI modules available on NAS resources are:

- mpi-intel/2018.3.222
- mpi-intel/2019.5.281
- mpi-intel/2020.0.166

For more information, see Modules.

Compiling Your Applications

Before you compile an application, load an Intel compiler module and an Intel MPI module. Ensure that no other MPI module (MPT or MVAPICH2) is loaded. For example:

```
% module purge
% module load mpi-intel/2020.0.166
% module load comp-intel/2020.4.304
```

Compile your application by running either the **`mpiifort`** script to invoke the ifort compiler or the **`mpiicc`** script to invoke the **`icc`** compiler. For example:

```
% mpiifort -o your_executable program.f
```

To view more information about the scripts, use the help option: **`mpiifort -help`** or **`mpiicc -help`**.

Running Your Applications

The following script provides an example of using the Hydra process manager command. You can replace **`mpiexec.hydra`** with **`mpiexec`** as the two binaries are identical.

```
## For applications that run the same number of
## MPI processes per node, use, for example
#PBS -lselect=4:ncpus=28:model=....

## If rank 0 needs all the memory in the
## first node, use, for example
#PBS -lselect=1:ncpus=1:model=...+3:ncpus=28...
..
module load mpi-intel/2020.0.166
module load comp-intel/2020.4.304    # or another Intel compiler module

...

# TOTAL_CPUS below represents value for the current job,
# it is to be replaced by an integer literal or
# by some variables that you set yourself

mpiexec.hydra -machinefile $PBS_NODEFILE -np TOTAL_CPUS your_executable
```

To view more mpiexec.hydra options, use the help option: **`mpiexec.hydra -help`**.

For more information about using Intel MPI libraries, see the Intel MPI Library Documentation website.

Porting with OpenMP

Building Applications

To build an OpenMP application, you need to use the **`-qopenmp`** Intel compiler flag:

```
%module load comp-intel/2020.4.304
%ifort -o your_executable -qopenmp program.f
```

Running Applications

The maximum number of OpenMP threads that an application can use on a compute node depends on:

- The number of physical processor cores in the node
- Whether hyperthreading is available and enabled

With hyperthreading, the operating system views each physical core as two logical processors, and can therefore assign two threads per core. This is beneficial only when one thread does not keep the functional units in the core busy all the time and can share the resources in the core with another thread. Running in this mode may take less than twice the wall time compared to running only one thread on the core.

Hyperthreading technology is available and enabled on all Pleiades, Electra, and Aitken Intel Xeon processor types. See the following table for the maximum number of threads possible on each processor type:

Pleiades Processor Type	Maximum Threads without Hyperthreading	Maximum Threads with Hyperthreading
Sandy Bridge	16	32
Ivy Bridge	20	40
Haswell	24	48
Broadwell	28	56
Skylake	40	80
Cascade Lake	40	80

TIP: Hyperthreading does not benefit all applications. Also, some applications may show improvement with some process counts but not with others, and there may be other unforeseen issues. Therefore, before using this technology in your production run, you should test your applications with and without hyperthreading. If your application runs more than twice as slow with hyperthreading than without, do not use it.
Here is sample PBS script for running OpenMP applications on a Pleiades Ivy Bridge node without hyperthreading:

```
#PBS -lselect=1:ncpus=20:ompthreads=20:model=ivy,walltime=1:00:00

module load comp-intel/2020.4.304

cd $PBS_O_WORKDIR

./your_executable
```

Here is sample PBS script with hyperthreading:

```
#PBS -lselect=1:ncpus=20:ompthreads=40:model=ivy,walltime=1:00:00
```

```
module load comp-intel/2020.4.304

cd $PBS_O_WORKDIR

./your_executable
```

Math & Scientific Libraries

Intel Math Kernel Library (MKL)

The Intel Math Kernel Library (MKL) is composed of highly optimized mathematical functions for engineering and scientific applications requiring high performance on Intel platforms. The functional areas of the library include linear algebra consisting of LAPACK and BLAS, fast Fourier transform (FFT), deep neural networks, vector statistics and data fitting, vector math, and miscellaneous solvers.

MKL is part of the Intel compiler releases. Once you load a compiler module (for example, **`module load comp-intel`**) the environment variable MKLROOT is automatically set and the path to the MKL library is automatically included in your default path.

A Layered Model for MKL

Intel employs a layered model for MKL. There are three layers: the interface layer, the threading layer, and the computational layer.

Interface Layer

The LP64 interface is for using 32-bit integer type and the ILP64 interface is for using 64-bit integer type.

Note: The SP2DP interface (used for mapping single-precision in Cray-style naming application and double-precision in MKL) has not been included in the MKL library releases since MKL 2017.

Threading Layers

Sequential

The sequential (non-threaded) mode does not require an OpenMP runtime library, and does not respond to the environment variable OMP_NUM_THREADS or its Intel MKL equivalents. In this mode, Intel MKL runs unthreaded code. However, it is thread-safe, which means that you can use it in a parallel region from your own OpenMP code.

You should use the library in the sequential mode only if you have a particular reason not to use Intel MKL threading. The sequential mode may be helpful when using Intel MKL with programs threaded with some non-Intel compilers or in other situations where you might need a non-threaded version of the library (for instance, in some MPI cases).

Note: The "sequential" library depends on the POSIX threads library (**`pthread`**), which is used to make the Intel MKL software thread-safe and should be listed on the link line.

Threaded

The "threaded" library supports the implementation of OpenMP and Threading Building Blocks (TBB), which Intel and GNU compilers provide. With the PGI compilers, only the OpenMP threading is available.

Computational Layer

For any given processor architecture (IA-32 or Intel 64) and operating system, this layer has only one computational library to link with, regardless of the interface and threading layer.

Compiler Support Runtime Libraries

Dynamically link the libiomp5 or libtbb library even if you link other libraries statically. Linking to the libiomp5 statically can be problematic, because the more complex your operating environment or application, the more likely redundant copies of the library are included. This may result in performance issues (oversubscription of threads) and even incorrect results. To link libiomp5 or libtbb dynamically, be sure the LD_LIBRARY_PATH environment variable is defined correctly.

Compiling and Linking with Intel MKL

Use the online Intel MKL Link Line Advisor to determine the libraries and options to specify on your link or compilation line.

For example, to do a dynamic linking and use parallel Intel MKL supporting LP64 interface for an OpenMP code, the MKL Link Line Advisor suggests:

```
 -L$/lib/intel64 -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm -ldl
```

The -mkl Switch

The Intel compiler provides an **`-mkl`** switch to link to certain parts of the MKL library. In many cases, using this switch (instead of having to explicitly specify the MKL libraries) is all you need.

```
-mkl[=]
          link to the Intel(R) Math Kernel Library (Intel(R) MKL) and
          bring in the associated headers
            parallel   - link using the threaded Intel(R) MKL libraries.
                         This is the default when -mkl is specified
            sequential - link using the non-threaded Intel(R) MKL libraries
            cluster    - link using the Intel(R) MKL Cluster libraries plus
                         the sequential Intel(R) MKL libraries
```

The libraries that are linked in for:

```
    * -mkl=parallel

          --start-group \
          -lmkl_intel_lp64 \
          -lmkl_intel_thread \
          -lmkl_core \
          -liomp5 \
          --end-group \

    * -mkl=sequential

          --start-group \
          -lmkl_intel_lp64 \
          -lmkl_sequential \
          -lmkl_core \
          --end-group \

    * -mkl=cluster

          --start-group \
          -lmkl_intel_lp64 \
          -lmkl_cdft_core \
          -lmkl_scalapack_lp64 \
          -lmkl_blacs_lp64 \
          -lmkl_sequential \
          -lmkl_core \
          -liomp5 \
          --end-group \
```

Note: Using **`--start-group`** and **`--end-group`** allows cycling through the enclosed libraries until all their inter-references (if any) have been resolved.

Where to find more information about MKL

The Intel Math Kernel Library Documentation includes HTML and PDF versions of the following guides:

- *Developer References for C and Fortran* - detailed function descriptions, including calling syntax
- *Developer Guides* - MKL usage information in greater detail

MKL FFTW Interface

FFTW is a free collection of C routines for computing the discrete Fourier Transform (DFT) in one or more dimensions, and provides portability across platforms. The Intel Math Kernel Library (MKL) offers FFTW2 (for version 2.*x*) and FFTW3 (for version 3.*x*) interfaces to the Intel MKL Fast Fourier Transform and Trigonometric Transform functionality. These interfaces enable applications using FFTW to gain performance with Intel MKL without changing the application source code.

Some users have installed FFTW in their own directories (for example, **`/u/username/fftw`**). If you choose to install these routines, you will link to the FFTW library as follows:

```
ifort -O3                          \
      -o fftw_example.exe fftw_example.f \
      -I/u/username/fftw/include         \
      -L/u/username/fftw/lib             \
      -lfftw2
or

ifort -O3                          \
      -o fftw_example.exe fftw_example.f \
      -I/u/username/fftw/include         \
      -L/u/username/fftw/lib             \
      -lfftw3
```

Note: The application programming interface of FFTW 3.*x* is incompatible with that of FFTW 2.*x*.

The MKL Interfaces

The MKL interfaces are available in the form of source files (under **`/nasa/intel/Compiler/[Version]/mkl/interfaces`**), which can be built into libraries and then linked when you compile your application.

Starting with Intel MKL release 10.2 (which was distributed in the Intel Compiler 11.1.*x* package), the FFTW3 interfaces are integrated into the MKL main libraries for ease of use. If you load an Intel compiler module (for example, **`comp-intel/2018.0.128`**), the above example of linking to an FFTW3 library in a user's directory can be changed to, for example:

```
module load comp-intel/2018.0.128

ifort -O3                                          \
      -o fftw_example.exe fftw_example.f                 \
      -I/nasa/intel/Compiler/2018.0.128/mkl/include/fftw \
      -mkl
```

TIP: The FFTW3 interfaces do not support **`long double`** precision because Intel MKL FFT functions operate only on single- and double-precision floating point data types. This means that the functions with prefix **`fftwl_`**, supporting the **`long double`** data type, are not provided. The interfaces have other limitations, as well. For more information, see the Intel Math Kernel Library documentation.
The older FFTW2 interfaces are not integrated into the MKL main libraries. For your convenience, we have built four interface libraries for every version of the Intel compiler:

- **`libfftw2xc_single_intel.a`**
- **`libfftw2xc_double_intel.a`**
- **`libfftw2xf_single_intel.a`**
- **`libfftw2xf_double_intel.a`**

Here, **`"single" and "double"`** refer to floating point numbers. These four libraries were built with Intel's default choice of 4-byte integer precision. So, the above example of linking to an FFTW2 library in a user's directory can be changed to the following, assuming single precision:

```
module load comp-intel/2018.0.128

ifort -O3                                             \
      -o fftw_example.exe fftw_example.f              \
      -I/nasa/intel/Compiler/2018.0.128/mkl/include/fftw  \
      -L/nasa/intel/Compiler/2018.0.128/mkl/lib/intel64   \
      -lfftw2xf_single_intel                          \
      -mkl
```

MPI Libraries

Using the NAS Recommended MPT Library

We recommend using HPE's Message Passing Toolkit (MPT) unless there is a strong reason for using another Message Passing Interface (MPI) implementation. HPE MPT is generally more efficient on HPE systems than other MPI libraries.

Benefits of Using HPE MPT

- HPE MPT contains MPI enhancements that are specific to HPE systems. In particular, it offers multiple features for scaling applications to very large process counts.
- The use of MPT on NAS systems is supported by HPE. MPT bugs and issues are tracked closely to provide timely resolution.

Using the Default MPT Version

Typically, there is more than one version of MPT library installed on NAS systems. You can access the currently recommended version by loading **`mpi-hpe/mpt`**, which points to the recommended version and environmental settings:

```
% module load mpi-hpe/mpt
```

Note: NAS administrators change the MPT version and environmental settings used by **`mpi-hpe/mpt`** from time to time, as needed, without notification. No recompilation or relinking should be needed for your application.

Loading **`mpi-hpe/mpt`** instead of a specific MPT module version provides multiple benefits:

- You always get the HPE MPT library with the version and environmental settings recommended by NAS staff.
- You don't have to change your script to load a specific MPT module when a new, improved version becomes available.
- Your script will not fail if an old MPT version is removed from the NAS systems.

If you would like to keep track of which MPT version and settings are being used by your application, you can set the following environmental variables:

```
(for csh, tcsh)
setenv MPI_VERBOSE            (shows MPT version and non-default settings)
setenv MPI_DISPLAY_SETTINGS   (shows all MPT settings)

(for sh, bash)
export MPI_VERBOSE=1
export MPI_DISPLAY_SETTINGS=1
```

In addition, we strongly recommend using the **`mpiexec`** launcher, instead of other launchers (such as **`mpirun`**) for your MPI application. The **`mpiexec`** launcher allows logging of the MPI environment used in your PBS job, such as the MPT version and settings, to the NAS system logfiles. If you notice a change in behavior of your jobs as a result of possible changes in **`mpi-hpe/mpt`**, please provide the PBS job ID when requesting help for investigation.

See Porting with HPE MPT for more information.

HPE MPT

HPE's Message Passing Interface (MPI) is a component of the HPE Message Passing Toolkit (MPT), a software package that supports parallel programming across a network of computer systems through a technique known as message passing. MPT requires the presence of an Array Services daemon (**arrayd**) on each host in order to run MPI processes.

HPE's MPT 1.*x* versions support the MPI 1.2 standard and certain features of MPI-2. The 2.*x* versions are fully MPI-2 compliant. Starting with version 2.10, MPT provides complete support for the new MPI 3.0 standard. Recent versions of MPT provide bug fixes and enhancementsâ such as an extension for the packet retry mechanism to cover more message types, and improvements to the congestion detection and control algorithmâ to address InfiniBand stability and scalability issues.

There may be more than one version of MPT on NAS systems. You can access the recommended version by running:

```
module load mpi-hpe/mpt
```

Note that certain environment variables are set or modified when an MPT module is loaded. To see what variables are set when you load **mpi-hpe/mpt**, complete these steps:

1. Run **module show mpi-hpe/mpt** to find out which version of MPT the module points to. For example:

   ```
   module show mpi-hpe/mpt
   /nasa/modulefiles/sles12/mpi-hpe/mpt:

   system              logger -p local2.info -t envmodules zsmith display mpi-hpe/mpt
   module-whatis    Loaded recommended version of MPT.
   module              load mpi-hpe/mpt.2.23
   ```
2. Run **module show *file_name*** to see the environment variables that are set for that version. For example:

   ```
   module show mpi-hpe/mpt.2.23
   ```

You can use the following commands to build an MPI application using HPE MPT:

```
%ifort -o executable_name prog.f -lmpi
%icc -o executable_name prog.c -lmpi
%icpc -o executable_name prog.cxx -lmpi++ -lmpi
%gfortran -I/nasa/hpe/mpt/2.23/include -o executable_name prog.f -lmpi
%gcc  -o executable_name prog.c -lmpi
%g++ -o executable_name prog.cxx -lmpi++ -lmpi
```

TIP: Note that the Fortran 90 **USE MPI** feature is supported for the **ifort** command, but not **gfortran**. Replace **USE MPI** with **include "mpif.h"** if you want to use **gfortran** to compile your Fortran 90 code and link to an HPE MPT library.

HPC-X MPI: An Alternative to MPT

HPC-X Message Passing Interface (MPI) is an implementation of Open MPI (based on version 4.x) that includes features and optimizations from Mellanox, the vendor of the InfiniBand networking hardware used in HECC systems. HPC-X MPI offers improved performance for some MPI routines, however, the main purpose we are supporting this implementation is to provide you with an alternative to using the HPE Message Passing Toolkit (MPT).

Some MPI constructs that had been removed from the MPI 3.0 standardâ   but had remained available for use in deprecated statusâ   were formally removed from OpenMPI-4.x. Our HPC-X MPI installation was built with a flag that makes these deprecated features visible again. For a list of these features, see Removed MPI Constructs on the Open MPI website.

Recommendation: Keep your MPI applications up to date with the current MPI standard.

To use HPC-X MPI on HECC systems, simply load the HPC-X module and your preferred Intel compiler suite, then recompile your application and its dependencies. Launch the newly built application as usual using the **mpiexec** launcher.

You can safely use the HPC-X build with any of the Intel compiler versions listed using command **module avail comp-intel**. The following example uses a specific Intel compiler:

```
% module purge
% module load comp-intel/2018.3.222 mpi-hpcx/2.4.0
```

The default process placement strategy is to place ranks round-robin by socket. This approach balances the computational load on the available hardware, but it might not be appropriate for all applications. You can force placement of ranks in linear core order by using the **mpiexec --map-by** option. For example:

```
% mpiexec -np XX --map-by ppr:YY:socket --bind-to core
```

where ***XX*** is the total number of ranks, ***YY*** is the user-specified number of ranks per socket, and the **bind-to** option is used to pin the ranks to specific cores.

Some workflow changes will be required if you manipulate $PBS_NODEFILE in order to launch multiple MPI jobs within a single PBS job. In this case, do the following:

- In the PBS resource request, include the maximum number of MPI ranks you intend to launch on each node.
- Create a separate rankfile for each invocation of **mpiexec**, using the following rankfile format:

  ```
  % rank XX=machineName slot=cpuID
  ```

  where ***XX*** is the MPI rank within the MPI sub-job, ***machineName*** is one of the names taken from the set provided by PBS (run **uniq $PBS_NODEFILE** to obtain this set), and ***cpuID*** is a hardware core ID (for core numbering, see the diagrams for each processor type: Cascade Lake, Skylake, Broadwell, Haswell, Ivy Bridge, Sandy Bridge).

This sample rankfile runs 16 ranks (0-15) evenly distributed on four sockets on two Ivy Bridge nodes (r469i0n10 and r469i0n11):

```
rank 0=r469i0n10 slot=0
rank 1=r469i0n10 slot=1
rank 2=r469i0n10 slot=2
rank 3=r469i0n10 slot=3
rank 4=r469i0n10 slot=10
```

```
rank 5=r469i0n10 slot=11
rank 6=r469i0n10 slot=12
rank 7=r469i0n10 slot=13
rank 8=r469i0n11 slot=0
rank 9=r469i0n11 slot=1
rank 10=r469i0n11 slot=2
rank 11=r469i0n11 slot=3
rank 12=r469i0n11 slot=10
rank 13=r469i0n11 slot=11
rank 14=r469i0n11 slot=12
rank 15=r469i0n11 slot=13
```

Optimizing/Troubleshooting

Debugging

Recommended Intel Compiler Debugging Options

Commonly Used Options for Debugging

-O*0*
: Disables optimizations. Default is **`-O2`**

-g
: Produces symbolic debug information in object file (implies **`-O0`** when another optimization option is not explicitly set)

-traceback
: Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at runtime.
: Specifying **`-traceback`** will increase the size of the executable program, but has no impact on runtime execution speeds.

-check all
: Checks for all runtime failures.
: Fortran only.

-check bounds
: Alternate syntax: **`-CB`**. Generates code to perform runtime checks on array subscript and character substring expressions.
: Fortran only.
: Once the program is debugged, omit this option to reduce executable program size and slightly improve runtime performance.

-check uninit
: Checks for uninitialized scalar variables without the *SAVE* attribute.
: Fortran only.

-check-uninit
: Enables runtime checking for uninitialized variables. If a variable is read before it is written, a runtime error routine will be called. Runtime checking of undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays.
: C/C++ only.

-ftrapuv
: Traps uninitialized variables by setting any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors. This option sets **`-g`**.

-debug all
: Enables debug information and control output of enhanced debug information. To use this option, you must also specify the **`-g`** option.

-gen-interfaces
-warn interfaces
: Tells the compiler to generate an interface block for each routine in a source file; the interface block is then checked with **`-warn`** interfaces.

Options for Handling Floating-Point Exceptions

-fpe{*0*|*1*|*3*}

Allows some control over floating-point exception (divide by zero, overflow, invalid operation, underflow, denormalized number, positive infinity, negative infinity or a NaN) handling for the main program at runtime.
Fortran only.
`-fpe`***0***: underflow gives 0.0; abort on other IEEE exceptions
`-fpe`***3***: produce NaN, signed infinities, and denormal results
Default is `-fpe`***3*** with which all floating-point exceptions are disabled and floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. Note that use of the default `-fpe`***3*** may slow runtime performance.

-fpe-all={*0*|*1*|*3*}

Allows some control over floating-point exception handling for each routine in a program at runtime. Also sets **`-assume ieee_fpe_flags`**. Default is `-fpe-all=`***3***.
Fortran only.

-assume ieee_fpe_flags

Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance.
Fortran only.

-ftz

Flushes denormal results to zero when the application is in the gradual underflow mode. This option has effect only when compiling the main program. It may improve performance if the denormal values are not critical to your application's behavior. Every optimization option O level, except `-O`***0***, sets **`-ftz`**.

Options for Handling Floating-Point Precision

-mp

Enables improved floating-point consistency during calculations. This option limits floating-point optimizations and maintains declared precision. `-mp`***1*** restricts floating-point precision to be closer to declared precision. It has some impact on speed, but less than the impact of **`-mp`**.

-fp-model precise

Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance.

-fp-model strict

Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

-fp-speculation=*off*

Disables speculation of floating-point operations. Default is `-fp-speculation=`***fast***

-pc{*64*|*80*}

For Intel EM64 only. Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the `-pc[`***n***`]` option. `-pc`***64*** sets internal FPU precision to 53-bit significand. `-pc`***80*** is the default and it sets internal FPU precision to 64-bit significand.

Common Causes of Segmentation Faults (Segfaults)

A segmentation fault (often called a *segfault*) can occur if a program you are running attempts to access an invalid memory location. When a segmentation fault occurs, the program will terminate abnormally with an error similar to the following message:

```
 SIGSEGV: Segmentation fault - invalid memory reference.
 forrtl: severe (174): SIGSEGV, segmentation fault occurred
```

The program may generate a core file, which can help with debugging.

If you use an Intel compiler, and you include the **`-g -traceback`** options, the runtime system will usually point out the function and line number in your code where a segmentation fault occurred. However, the location of the segmentation fault might not be the root problemâ   a segfault is often a symptom, rather than the cause of a problem.

Common Segfault Scenarios

Common scenarios that can lead to segmentation faults include running out of stack space and issues resulting from bugs in your code.

Running Out of Stack Space

Stack space is a segment of program memory that is typically used by temporary variables in the program's subroutines and functions. Attempting to access a variable that resides beyond the stack space boundary will cause segmentation faults.

The usual remedy is to increase the stack size and re-run your program. For example, to set the stack size to unlimited, run:

For csh
```
unlimit stacksize
```
For bash
```
ulimit -s unlimited
```

On the Pleiades front-end nodes (PFEs), the default stack size is set to 300,000 kilobytes (KB). On the compute nodes, PBS sets the stack size to **`unlimited`**. However, if you use **`ssh`** to connect from one compute node to another (or several others) in order to run programs, then the stack size on the other node(s) is set to 300,000 KB.

Note: Setting the stack size to **`unlimited`** on the PFEs might cause problems with Tecplot. For more information, see Tecplot.

Bugs in Your Fortran Code

In Fortran programs, the most common bugs that cause segmentation faults are array bounds violationsâ   attempts to write past the declared bounds of an array. Occasionally, uninitialized data can also cause segmentation faults.

Array Bounds Violations

To find array bounds violations, re-run your code with the Intel ifort compiler using the **`-check`** (or **`-check all`**) option in combination with your other compiler options. When you use the **`-check`** option, the Fortran runtime library will flag occurrences of array bounds violations (and some other programming errors).

When the runtime library encounters the first array bounds violation, it will halt the program and provide an error message indicating where the problem occurred. You may need to re-run the code multiple times if there is more than one array bounds violation.

Note: Code compiled with the **`-check`** option may run significantly slower than code compiled with normal optimization (without the **`-check`** option).

Uninitialized Variables

You can use the **`-init=keyword`** option (available in the 2015 Intel Fortran compiler and later versions) to check uninitialized variables. The following keywords can be used with the **`-init`** option:

`[no]arrays`
: Determines whether the compiler initializes variables that are arrays or scalars. Specifying **`arrays`** initializes variables that are arrays or scalars. Specifying **`noarrays`** initializes only variables that are scalars. You must also specify either **`init snan`** or **`init zero`** when you specify **`init [no]arrays`**.

`[no]snan`
: Determines whether the compiler initializes to signaling NaN all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated.

`[no]zero`
: Determines whether the compiler initializes to zero all uninitialized variables of intrinsic type REAL, COMPLEX, INTEGER, or LOGICAL that are saved, local, automatic, or allocated.

Note: The **`-init`** compiler option does not catch all possible uninitialized variables. To find more, you can use the NAS-developed **`uninit`** tool. For information about using this tool, see the NAS training presentation uninit: Locating Use of Uninitialized Data in Floating Point Computation in Big Applications.

For more information about segmentation faults, see:

- Determining Root Cause of Segmentation Faults SIGSEGV or SIGBUS errors (Intel Developer Zone)
- Segmentation Fault (Wikipedia)

TotalView

TotalView is a GUI-based debugging tool that provides control over processes and thread execution, as well as visibility into program state and variables, for C, C++ and Fortran applications. It also provides memory debugging to detect errors such as memory leaks, deadlocks, and race conditions. You can use TotalView to debug serial, OpenMP, or MPI codes.

Using TotalView to Debug Your Code

To find out which versions are available as modules, use the **`module avail`** command.

Before You Begin

Our current licenses allow using TotalView for up to a total of 256 processes. Use the following command to find out whether there are unused licenses before you start TotalView:

```
pfe% /u/scicon/tools/bin/check_licenses -t
```

Ensure that X11 forwarding is set up on your system. Alternately, you can use the **`ssh -X`** or **`-Y`** options to enable X11 forwarding for your SSH session.

Note: If you are using a NAS-supported workstation or compute server, X11 forwarding should already be set up on your system. If the response of the GUI via X11 forwarding is slow, you will need to set up a VNC session.

Complete these steps:

1. Compile your program using the **`-g`** option.
2. Start a PBS session.

   On Pleiades, Aitken, and Electra:

   ```
   % qsub -I -X -q devel -lselect=2:ncpus=8:model=xxx,walltime=1:00:00
   ```

   where xxx is one of the following: **`san, ivy, has, bro, sky_ele, cas_ait`**, or **`rom_ait`**.

   On Endeavour3/4:

   ```
   % qsub -I -X  -lselect=1:ncpus=28:mem=185GB:model=cas_end,walltime=1:00:00 \
   -q queue_name@pbspl4
   ```

   Note: The command line above is too long to be formatted as one line, so it is broken with a backslash (\).
3. Test the X11 forwarding with **`xclock:`**

   ```
   % xclock
   ```

Debugging with TotalView

Load the module:

```
% module load totalview/2017.0.12
```

The method you use to run TotalView depends on the application you want to debug.

For Serial Applications

Launch TotalView by running the **`totalview`** command and specifying the application:

```
% totalview ./a.out
```

Or, if your application requires arguments:

```
% totalview ./a.out -a arg_1 arg_2
```

For MPI Applications Built with HPE MPT

- For older versions of MPT, such as mpt.2.17r13 or mpt.2.21, use the **`-tv`** option of **`mpiexec_mpt`**, as shown in this example:
    1. Load the MPT module:

       ```
       % module load comp-intel/2018.3.222
       % module load mpi-hpe/mpt.2.17r13
       ```
    2. Launch your program as follows:

       ```
       % mpiexec_mpt -tv -np 16 ./a.out
       ```

  TIP: If you want to use the ReplayEngine feature of TotalView, you need to set these two environment variables:

  ```
  setenv IBV_FORK_SAFE 1
  setenv LD_PRELOAD /nasa/totalview/toolworks/totalview.2017.0.12/ \
  Â     linux-x86-64/lib/undodb_infiniband_preload_x64.so
  ```

  Note that the second command line above is too long to be formatted as one line, so it is broken with a backslash (\).
- For newer versions of MPT where the **`-tv`** option is no longer supported, such as mpt.2.23 or mpt.2.25, use the following method:
    1. Load the latest MPT module:

       ```
       % module load comp-intel/2018.3.222
       % module load mpi-hpe/mpt
       ```
    2. Launch your program as follows:

       ```
       % totalview mpiexec_mpt.real -a -np 16 ./a.out
       ```

       If the MPI launcher fails to launch the executable and recommends setting **`MPI_SHEPHERD=true`**, then set that environment variable and try running TotalView again.

For more information, see the TotalView documentation.

GNU Debugger (GDB)

The GNU Debugger (GDB) is available on HECC systems in the **`/usr/bin`** directory. GDB can be used to debug programs written in C, C++, Fortran, and Modula-a.

GDB can perform four main tasks:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Be sure to compile your code with **`-g`** for symbolic debugging.

GDB is typically used in the following ways:

- Start the debugger by itself:

  ```
  % gdb
  (gdb)
  ```
- Start the debugger and specify the executable:

  ```
  % gdb your_executable
  (gdb)
  ```
- Start the debugger, and specify the executable and core file:

  ```
  % gdb your_executable core-file
  (gdb)
  ```
- Attach **`gdb`** to a running process:

  ```
  %gdb your_executable pid
  (gdb)
  ```

At the prompt (**`gdb`**), enter commands such as **`break`** (for setting a breakpoint), **`run`** (to start running your executable, and **`step`** (for stepping into next line). Read **man gdb** to learn more on using **`gdb`**.

Finding Hotspots in Your Code with the Intel VTune Command-Line Interface

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Pleiades, Aitken, and Electra Intel processor types: Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the **`-g`** option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, such as **`mpi-hpe/mpt.2.25`**, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

   ```
   module load vtune/2021.9
   ```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the **`vtune`** command line that is appropriate for your code, as listed below. Use the **`-collect`** (or **`-c`**) option to run the hotspots collection and the **`-result-dir`** (or **`-r`**) option to specify a directory.

Note: The **`vtune`** command replaces **`amplxe-cl`**, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs
```

Data collected by VTune for the **`a.out`** application are stored in the **`r000hs`** directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in the **`r000hs`** directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir
```

Note: If your **`vtune`** run fails with **`MPT ERROR`** and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default to **`stdout`**. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, **`r000hs.r587i0n0`** and **`r000hs.r587i0n1`**, where each directory contains 24 subdirectories, (**`data.[`*`0-23`*`]`**).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hsÂ  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), **`vtune-gui`**. Since the Pleiades front-end systems (PFEs) do not have sufficient memory to run **`vtune-gui`**, run it on a compute node as follows:

1. On a PFE, run **`echo $DISPLAY`** to find its setting (for example, **`pfe22:102.0`**).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session
   (**`qsub -I -X`**) or a reservable front end (**`pbs_rfe`**).
5. On the compute node, start **`vtune-gui`** with name of the result directory, for example, **`r000hs`**.

   ```
   compute_node% module load vtune/2021.9
   compute_node% vtune-gui r000hs
   ```

Generating a Report

You can also use the **`vtune`** command with the **`-report`** option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to **`stdout`**, or to a specific output text file, such as **`output.txt:`**

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```
- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```
- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```
- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```
- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Using Gprof for Performance Analysis

Gprof is a compiler-assisted performance profiler for C, Fortran, and Pascal applications running on Unix systems. You can use Gprof to help identify hotspots in your application where code optimization efforts may be most useful.

Gprof uses a hybrid of sampling and instrumentation, and provides the following information:

- The number of calls to each function and the amount of time spent there.
- Information about the caller-callee relationship.

Note: Gprof only measures the user code; it does not provide information on time spent in the kernel (such as system calls or I/O wait time).

The profiling data will be collected in a file called **`gmon.out`**, which will be generated at the end of a successful, uninterrupted run.

Gprof is available in the **`/usr/bin`** directory on Pleiades. To use this tool, follow the instructions in the sections below.

Compiling and Linking to Enable Profiling with Gprof

To enable profiling with Gprof, add one of the options shown below when you compile your code:

- With Intel compilers, add the **`-p`** option (alternatively you can add **`-pg`**, which is deprecated, but still works).
- With PGI compilers, add the **`-pg`** option.
- With GNU compilers, add the **`-pg`** option. You might also have to use the **`-O0`** option, if you do not get meaningful profiling results when using higher levels of optimization.

Collecting Profiling Data

To collect profiling data, simply run your gprof-enabled executable the same way you would run a non-gprof-enabled executable. The data will be collected in a file called **`gmon.out`**.

OpenMP Applications

If you are using an OpenMP application, gprof does not generate per-thread profiling data. Only one **`gmon.out`** file is produced.

Note: If a file named **`gmon.out`** already exists in the directory, it will be overwritten.

MPI Applications

For MPI applications, if you use the Intel MPI library, no additional steps are required before you run your code. However, if you use the HPE MPT library, you also need to set the MPI_SHEPHERD variable as follows before you run the code; otherwise, the timing information will not be shown:

```
export MPI_SHEPHERD=true (bash)
setenv MPI_SHEPHERD true (csh)
```

Each MPI process will generate a profile with the same filename, **`gmon.out`**. These files will overwrite one another when they are written to a global filesystem. Therefore, to avoid this behavior and produce a profile with a distinct filename for each process, do:

```
export GMON_OUT_PREFIX=gmon.out (bash)
setenv GMON_OUT_PREFIX gmon.out (csh)
```

For example:

```
#PBS ...

module load comp-intel/2020.4.304
module load mpi-hpe/mpt.2.25
export MPI_SHEPHERD=true
export GMON_OUT_PREFIX=gmon.out

mpiexec a.out
```

This operation will generate a file called **`gmon.out.`*`pid`***, where ***`pid`*** is the process ID of an MPI process. With *N* ranks, you should get *N* such files, with filenames differing only in their ***`.pid`*** extensions. You can then analyze an individual **`gmon.out.`*`pid`*** file or several at the same time.

Generating ASCII Gprof Output

You can use the **`gprof`** command to convert binary data in **`gmon.out`** into a human-readable format.

```
gprof [options] executable_name [gmon.out] [ > analysis.output ]
```

There are two types of output: *flat profile* and *call graph.*

The flat profile shows how much time your program spent in each function, and how many times that function was called. This profile helps to identify hotspots in your application. Hotspots are shown at the top of the flat profile.

The call graph shows, for each function, which functions called it; which other functions it called; and how many times the calls occurred. The call graph also provides an estimate of how much time was spent in the subroutines of each function, which can suggest places where you might try to eliminate function calls that use a lot of time.

Commonly Used gprof Command Options

Some common options are described here. Read **man gprof** for more information.

- **`-p`** prints a flat profile. For example:

  ```
  gprof -p a.out gmon.out.1001 gmon.out.1002
  ```
- **`-q`** prints a call graph. For example:

  ```
  gprof -q a.out gmon.out.*
  ```
- **`-s`** sums up the information from multiple profiling data files and produces a file called **`gmon.sum`** for analysis. For example:

  ```
  gprof -s a.out gmon.out.*
  gprof a.out gmon.sum > analysis.output
  ```
- **`-b`** omits verbose texts that explain the meaning of all of the fields in the tables.

Performance Analysis

Getting a Quick Performance Overview with Intel APS

The Intel Application Performance Snapshot (APS) tool provides a quick overview of your application's performance on processor and memory usage, message passing interface (MPI), and I/O, as well as load imbalance between threads or processes. In addition to the performance metrics listed below, APS also provides suggestions for performance enhancement opportunities and additional Intel profiling tools you can use to get more in-depth analysis.

Note: Although the Intel APS tool can be used for any of the Pleiades, Aitken, and Electra Intel Xeon processor types, it does not fully support the Sandy Bridge and Haswell processor types. APS cannot be used for the Aitken Rome processors.

Before You Begin

Note the following information:

- The Intel C/C++ or Fortran Compiler is not required, but is recommended. However, analysis of OpenMP imbalance is only available for applications that use the Intel OpenMP runtime library.
- There is no support for OpenMPI.
- Analysis of MPI imbalance is only available when you are using the Intel MPI Library version 2017 and later. (With the exception of the MPI imbalance metric, APS MPI analysis works for applications that use the NAS-recommended HPE MPT library.)

TIP: The NAS-developed MPIProf tool provides similar MPI analysis with MPI imbalance information included.

Running an APS Analysis

On Pleiades, APS is available via the **`vtune/2021.9`** module. See **`aps --help`** for available APS options.

To run an analysis for serial and OpenMP applications:

```
module load vtune/2021.9
aps <options> ./a.out
```

To run an analysis for MPI applications:

```
module load vtune/2021.9

module load mpi-hpe/mpt.2.25

setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np xx aps <options> ./a.out
```

TIP: The process/thread pinning tools **`mbind.x`** and **`omplace`** can be used with **`aps`**. Either of the following two methods may work:

- **`mpiexec -np xx aps mbind (or omplace) <mbind or omplace options> a.out`**
- **`mpiexec -np xx mbind (or omplace) <mbind or omplace options> aps <options> a.out`**

At the end of the run, APS provides a directory that contains the data collected during the analysis. The default name of the directory is **`aps_result_`*`yyyymmdd`***.

WARNING: The directory will be overwritten if you run another APS analysis in the same directory on the same day.

To send the results to another directory, you can include **`--result-dir=`*`dir_name`*** or **`-r=`*`dir_name`*** option in the **`aps`** command line (where **`dir_name`** represents the directory name of your choice).

Viewing the Report

APS provides a text summary and an HTML file named **`aps_report_`*`yyyymmdd_hhmmss`*`.html`** (where **`yyyymmdd_hhmmss`** is the date and time when the HTML file is created). The HTML file contains the same information as the text summary, but also offers the following features when you open the file with a web browser:

- A description of each metric is shown when you hover your mouse over the metric name.
- Possible performance issues of your run are highlighted in red.
- Suggestions with links to Intel tools are provided to help with performance enhancement.

Note: NAS security policy prohibits using web browsers on Pleiades. To view the HTML file, transfer it to your own workstation.

For Serial and OpenMP Applications

For serial or OpenMP applications, no action is required to generate the reports. At the end of the run, APS automatically generates the text summary (appended to your application's **`stdout`**) and the HTML file.

For MPI Applications

For MPI applications, you must generate the text summary (shown on your screen) and HTML file after the analysis completes, as follows:

```
aps --report=dir_name
```

where **`dir_name`** is the name of the directory created at the end of the analysis run.

If the summary report shows that your application is MPI-bound, run the following command to get more details about message passing, such as message sizes, data transfers between ranks or nodes, and time in collective operations:

```
aps-report <options> dir_name
```

See **`aps-report --help`** for available options.

You can also rerun the analysis after setting additional environmental variables. For example:

```
setenv MPS_STAT_LEVEL n
or
setenv APS_STAT_LEVEL n
```

where *n* is 2, 3 or 4 to get more detailed information on your application's MPI performance.

Note: If MPS_STAT_LEVEL (or APS_STAT_LEVEL) is set to 3 or 4, you must use mpi-hpe/mpt.2.18r160 or a later version when you run the analysis to avoid a segmentation fault (SEGV).

Quick Metrics Reference

Intel APS collects the metrics described in the following list.

Note: These descriptions are adapted from the Application Performance Snapshot User's Guide for Linux OS, where you can find additional details about each metric.

Elapsed Time
: Execution time of specified application in seconds.

SP GFLOPS
: Number of single precision giga-floating point operations (gigaflops) calculated per second. SP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

DP GFLOPS
: Number of double precision giga-floating point operations calculated per second. DP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

Cycles per Instruction Retired (CPI) Rate
: The amount of time each executed instruction took measured by cycles. A CPI of 1 is considered acceptable for high performance computing (HPC) applications, but different application domains will have varied expected values. The CPI value tends to be greater when there is long-latency memory, floating-point, or SIMD operations, non-retired instructions due to branch mis-predictions, or instruction starvation at the front end.

CPU Utilization
: Helps evaluate the parallel efficiency of your application. Estimates the utilization of all the logical CPU cores in the system by your application. 100% utilization means that your application keeps all the logical CPU cores busy for the entire time that it runs. Note that the metric does not distinguish between useful application work and the time that is spent in parallel runtimes.

MPI Time
: Time spent inside the MPI library. Values more than 15% might need additional exploration on MPI communication efficiency. This might be caused by high wait times inside the library, active communications, non-optimal settings of the MPI library. See MPI Imbalance metric to see if the application has load balancing problem.

MPI Imbalance
: Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data.

Serial Time
: Time spent by the application outside any OpenMP region in the master thread during collection. This directly impacts application Collection Time and scaling. High values might signal a performance problem to be solved via code parallelization or algorithm tuning.

OpenMP Imbalance
: Indicates the percentage of elapsed time that your application wastes at OpenMP synchronization barriers because of load imbalance.

Memory Stalls
: Indicates how memory subsystem issues affect the performance. Measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This metric's value can indicate that a significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. See the second level metrics to define if

the application is cache- or DRAM-bound and the NUMA efficiency.

Cache Stalls
: indicates how often the machine was stalled on L1, L2, and L3 cache. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data.

DRAM Stalls
: Indicates how often the CPU was stalled on the main memory (DRAM) because of demand loads or stores.

DRAM Bandwidth
: The metrics in this section indicate the extent of high DRAM bandwidth utilization by the system during elapsed time. They include:

  - ◊ **Average Bandwidth:** Average memory bandwidth used by the system during elapsed time.
  - ◊ **Peak:** Maximum memory bandwidth used by the system during elapsed time.
  - ◊ **Bound:** The portion of elapsed time during which the utilization of memory bandwidth was above a 70% threshold value of the theoretical maximum memory bandwidth for the platform.

  Some applications can execute in phases that use memory bandwidth in a non-uniform manner. For example, an application that has an initialization phase may use more memory bandwidth initially. Use these metrics to identify how the application uses memory through the duration of execution.

NUMA: % of Remote Accesses
: In non-uniform memory architecture (NUMA) machines, memory requests missing last level cache may be serviced either by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than those to local DRAM. It is recommended to keep as much frequently accessed data local as possible. This metric indicates the percentage of remote accesses, the lower the better.

Vectorization
: The percentage of packed (vectorized) floating point operations. The higher the value, the bigger the vectorized portion of the code. This metric does not account for the actual vector length used for executing vector instructions. As a result, if the code is fully vectorized, but uses a legacy instruction set that only utilizes a half of the vector length, the Vectorization metric is still equal to 100%.

Instruction Mix
: This section contains the breakdown of micro-operations by single precision (SP FLOPs) and double precision (DP FLOPs) floating point and non-floating point (non-FP) operations. SP and DP FLOPs contain next level metrics that enable you to estimate the fractions of packed and scalar operations. Packed operations can be analyzed by the vector length (128, 256, 512-bit) used in the application.

  - ◊ **FP Arith/Mem Rd Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory read instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution.
  - ◊ **FP Arith/Mem Wr Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution. The metric value might indicate unaligned access to data for vector operations.

Additional Resources

- Getting Started with Application Performance Snapshot - Linux OS
- Application Performance Snapshot User's Guide for LinuxOS

Using Intel Advisor for Better Threading and Vectorization

You can use Intel Advisor to identify issues and help you with threading and vectorization optimization on the specific Intel processors that your Fortran, C, or C++ applications run on. The software, which is available as a module on HECC systems, provides recommended workflows for these two optimization areas.

Intel Advisor offers both a command-line interface (CLI) and a graphical user interface (GUI). We recommend that you use the CLI (**`advixe-cl`**) in a PBS batch job to collect data, and then use the GUI (**`advixe-gui`**) to analyze the result, as it provides a more comprehensive view than the CLI. However, please note that learning how to navigate the GUI will take some effort and time. For more information, see the Intel documentation, Intel Advisor GUI.

Analysis Types

Each Intel Advisor workflow involves a combination of some of the analysis types listed below. For detailed information about these workflows, see the Intel documentation, Getting Started with Intel Advisor, or access the CLI help page by running **`advixe-cl -h workflow`**.

Survey analysis
: Helps to identify loop hotspots and provides recommendations for how to fix vectorization issues.

Trip counts and flops analysis
: Counts the number of times loops are executed and provides data about floating-point operations (flops), memory traffic, and AVX-512 mask usage. This analysis requires more instrumentation and will perturb your application more. Thus, it takes longer time than the survey analysis.

Dependencies analysis
: Checks for real data dependencies in loops the compiler did not vectorize because of assumed dependencies.

Memory access patterns analysis
: Checks for memory access issues such as non-contiguous, or non-unit stride access.

Suitability analysis
: Predicts the maximum speed-up of your application, based on the inserted annotations and a variety of what-if modeling parameters you can experiment with. Use this information to choose the best candidates for parallelization with threads.

Roofline analysis
: Runs two analyses one by one: (1) the survey analysis, and (2) the trip counts and flops analysis. A roofline chart is then generated which plots an application's achieved floating-point performance and arithmetic intensity (AI) against the machine's maximum achievable performance.

: Note: For more information about running this analysis, see Roofline Analysis with Intel Advisor.

Compiling Your Code

At the minimum, you should include the following options when you compile your code:

```
-g -O2 (or higher)
-g -O2 (or higher) -qopenmp  (for OpenMP applications)
```

Depending on the processors your application runs on, you may consider adding an option that instructs the compiler to generate a binary that targets certain instruction sets. The options are listed in the following table.

Processor Type	Option
Skylake	-xCORE-AVX512
Broadwell, Haswell	-xCORE-AVX2
IvyBridge, SandyBridge	-xAVX
Multiple auto-dispatch code paths	-axCORE-AVX512,CORE-AVX2,AVX -xSSE4.2

Collecting Data

Load the module and collect data:

```
module load advisor/2018
advixe-cl -collect=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

where:

- **`<string>`** is one of the following analysis types: **`survey, tripcounts, dependencies, map,`** or **`suitability`**
  Note: For non-MPI applications, **`<string>`** can also be **`roofline`**. For MPI applications, roofline analysis can only be done by performing **`survey`** and **`tripcounts`** in two separate steps.
- **`<target>`** is your executable.

For example:

```
advixe-cl -collect survey -project-dir my_result -- ./a.out
```

For information about **`action-option`** and **`global-option`**, run **`advixe-cl -h collect`**.

Analyzing the Result

The data collected are stored in a directory tree (for example, **`my_result`**), which you specify using an **`action-option`** called **`-project-dir`**.

Depending on the analysis type used in data collection, you might find the subdirectories **`hsxxx, trcxxx, dpxxx, and mpxxx`** in the **`my_result/e000`** directory (for serial or pure OpenMP applications) or in the **`my_result/rank.n`** directory (for the nth-rank of an MPI application).

To view the result on a Pleiades front end system (PFE), use either the Intel Advisor GUI (**`advixe-gui`**) or the CLI (**`advixe-cl).`**

Use the Intel Advisor GUI

Load the module and run **`advixe-gui`**:

```
module load advisor/2018
advixe-gui my_result
```

After the GUI starts, click **Show My Result**. In the main window, you can choose one of three panels: **Summary** (default view), **Survey & Roofline**, or **Refinement Reports**. All three are described below.

Summary
: Might show gigaflops count, AI, number of threads, loop metrics, vectorization gain/efficiency, top time-consuming loops, and platform information.

Survey & Roofline
: Might show the roofline chart, source code, assembly code, detailed info on flops, AI, efficiency, reason for no vectorization of the top-time-consuming loops.

Refinement Reports
: Might show memory access patterns report (1-stride, 0-stride, constant-stide, irregular stride) and dependencies report.

Use the Intel Advisor CLI

Load the module and run **`advixe-cl`**:

```
module load advisor/2018
advixe-cl -report=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

For information on **`<string>`**, **`action-option,`** and **`global-option,`** run: **`advixe-cl -h report`**

Note: If the job runs too quickly, there is not enough data collected to generate a report.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor CLI
- Intel Advisor GUI
- Intel Advisor presentation by Intel engineer Jackson Marusarz
- Transforming Serial Code to Parallel Using Threading and Vectorization Part 1 and Part 2.

Running a Roofline Analysis with Intel Advisor

A roofline analysis helps you determine whether your application has achieved the best possible performance, limited by the machine capabilities. If not, you can explore the possibility of algorithm changes to improve performance.

Running the Analysis

The Intel Advisor tool is available as a module on HECC resources. To do a roofline analysis, load the module and run two analyses: survey and tripcounts, as follows:

```
module use /nasa/modulefiles/testing
module load mpi-hpe/mpt.2.18r160
module load advisor/2018
mpiexec -np x advixe-cl -collect survey -project-dir my_result -- ./a.out
mpiexec -np x advixe-cl -collect tripcounts -flop -project-dir my_result -- ./a.out
```

Note: Be sure to specify the same directory (**`-project-dir`**) for the survey and tripcounts analyses. The tripcounts run will take longer time than the survey run.

If these two runs are successful, you can start the Intel Advisor GUI and choose one of the ranks to analyze.

Reviewing the Results

To start the GUI:

```
advixe-gui my_result
```

To see a roofline chart for the rank chosen, do one of the following:

- In the navigation panel, click the black icon (black-icon.png) under **Run Roofline**.
- Click the **Survey & Roofline** panel in the main window, then click the vertical bar labeled **ROOFLINE**.

In the sample roofline chart shown below, the X axis is arithmetic intensity (measured in flops/byte) and the Y axis is the performance in Gflops/second, both in logarithmic scale:

roofline_example(1).png

Note: This sample chart is reproduced from the Intel documentation, Getting Started with Intel Advisor.

Before it collects data on your application, Intel Advisor runs benchmarks to measure the hardware limitations of your machine. It plots these on the chart as lines, called *roofs*. The horizontal lines represent the number of floating point computations of a given type your hardware can perform in a given span of time. The diagonal lines are representative of how many bytes of data a given memory subsystem can deliver per second.

In the chart, each dot is a loop or function in your application. The dot's position indicates the performance of the loop or function, which is affected by its optimization and its arithmetic intensity. The dot's size and color indicate how much of the total application time the loop or function takes. In the sample chart shown above, loops A and G (large red dots), and to a lesser extent loop B (yellow dot far below the roofs), are the best candidates for optimization. Loops C, D, and E (small green dots) and H (yellow dot) are poor candidates because they do not have much room to improve.

Important: Intel Advisor uses a cache-aware roofline model. In the classic roofline model, a kernel's arithmetic intensity would change with problem size or cache usage optimization, because the byte count was based on DRAM traffic only. This is not the case in the cache-aware roofline model, where arithmetic intensity is a fixed value tied to the algorithm itself; it only changes when the algorithm is altered, either by the programmer or occasionally by the compiler.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor Roofline
- Getting Started with Intel Advisor Roofline Feature
- Intel video: Introduction to Intel Advisor Roofline Feature
- Roofline: An Insightful Visual Performance Model for Floating-Point Programs and

Multicore Architecture, 2009 (PDF)
- Cache-aware Roofline Model: Upgrading the Loft, 2013 (PDF)

Using MPIProf for Performance Analysis

MPIProf is a lightweight, profile-based application performance analysis tool that works with many MPI implementations, including HPE MPT. The tool gathers statistics in a counting mode via PMPI, the MPI standard profiling interface.

MPIProf can report profiling information about:

- Collective and/or point-to-point MPI functions called by an application (time spent, number of calls, and message size)
- MPI and POSIX I/O statistics
- Memory used by processes on each node
- Call path information of MPI calls (requires the **`-g`** compiler option and the **`-cpath`** mpiprof option)

There are two ways you can use MPIProf:

- The **`mpiprof`** profiling tool on the command line:

  ```
  mpiexec -n <N> mpiprof [-options] [-h|-help] a.out [args]
  ```
- MPIProf API routines

mpiprof Usage Example

Because the command-line method does not require changing or recompiling your application, we recommend this method for general use. To run **`mpiprof`**, load the modulefiles for your compiler, MPI library, and the **`mpiprof`** tool, then run the command line. For example:

```
module load comp-intel/2015.3.187 mpi-hpe/mpt
module load /u/scicon/tools/modulefiles/mpiprof
mpiexec -n 128 mpiprof -o mpiprof.out a.out
```

Note: If the **`-o`** option is not included, the profiling results are written to the **`mpiprof_stats.out`** file, or the <**`a.out`**>**`_mpiprof_stats.out`** file if the executable name is known.

For more details about using this tool, see the MPIProf user guide in the **`/u/scicon/tools/`** directory on Pleiades:

`/u/scicon/tools/opt/mpiprof/default/doc/mpiprof_userguide.pdf`

MPIProf was developed by NAS staff member Henry Jin.

Using MPInside for Performance Analysis and Diagnosis

MPInside, an MPI application performance analysis and diagnostic tool from HPE, is a lightweight program that can be used with thousands of ranks. The tool is profile-based via PMPI, the MPI standard profiling interface. An MPInside analysis can provide you with communication statistics (collective and point-to-point) as well as information that can help you diagnose communication issues such as:

- Slow MPI communications caused by non-synchronous send/receive pairs
- Load imbalance (compute time or message bandwidth) among ranks
- Reasons a code runs well with one MPI library or one platform, but not another

In addition to MPI communication profiling, MPInside provides the following optional features:

- MPI performance modeling
- I/O measurement

MPInside supports the full MPI 2.2 standard and the commonly used MPI libraries: HPE MPT, Intel MPI, and OpenMPI.

Note: Some features may be limited to HPE MPT.

Basic Usage

Most features of MPInside do not require instrumentation, recompiling, or relinking of your application. However, using the **`-g`** compiler flag is recommended.

The basic MPInside measurement produces a flat profile. No information is provided about the source lines that make the MPI calls.

Environment Setup

To set up your environment to use MPInside, load the MPInside module and an MPI library module. Set the **`MPINSIDE_LIB`** environment variable to match the MPI library module. For example:

```
module load MPInside/3.6.2

module load mpi-hpe/mpt
setenv MPINSIDE_LIB MPT (or IMPI, or OPENMPI)
```

Note: **`MPINSIDE_LIB`** defaults to **`MPT`**.

Communication Profiling

To perform an analysis using MPInside, run the following commands (where *XX* = the number of MPI processes):

```
mpiexec -np XX MPInside a.out
```

Note: The **`MPInside`** command is case-sensitive.

By default, MPInside measures the elapsed times of the MPI functions and collects message statistics. After the MPI application completes successfully, the measurement and the statistics data are reported in a text file called mpinside_stats. This output file contains five sections:

- The elapsed time of each function in seconds (broken down into columns for computation time and for various MPI function times, with one rank per row)
- Total megabytes (MBytes) sent by each rank
- Number of "send" calls by each rank
- Total MBytes received by each rank
- Number of "recv" calls by each rank

You can import the mpinside_stats file to an Excel spreadsheet and plot the data for visual analysis (for example, analysis of possible load imbalance or communication characteristics). You can also compute derived metrics, such as average message size, from the mpinside_stats file.

Additional Features

MPInside uses environment variables to set all of its options. These optional variables are documented in the **mpinside** man page.

Recommended Features

We recommend the following settings:

`MPINSIDE_LITE=1`
  Use when synchronization frequency is very high
`MPINSIDE_CUT_OFF=0.0`
  Do not cut off reporting of any MPI functions
`MPINSIDE_OUTPUT_PREFIX={run i.d.}`
  Change default report file name
`MPINSIDE_PRINT_ALL_COLUMNS=1`
  Enable reporting of all columns (even if the values are zero) to allow easier comparisons between runs
`MPINSIDE_PRINT_SIZ_IN_K=1`
  Print in KB units instead of MB

Advanced Features

The advanced features described below may also be useful.

MPINSIDE_SHOW_READ_WRITE

Measures the I/O time and includes several columns in the mpinside_stats output file indicating the time, number of Mbytes, and number of calls associated with the **`libc`** I/O functions. Note that the MPI function times and I/O times are subtracted from the total elapsed run times, and the result is reported as "Comput" time.

MPINSIDE_SIZE_DISTRI [T+]nb_bars[:*first-last*]

Prints a histogram of the request sizes distribution at the end of mpinside_stats for each rank specified in [:*first-last*]. If T+ is specified, size distribution time histograms are also printed. For example, setting MPINSIDE_SIZE_DISTRI to T+12:0-47 will print two histograms, as follows, for each rank (from 0 to 47) at the end of mpinside_stats:

1. Size distribution for sizes 0, 32, 64, 128, 256, 512, ..., up to 65536
2. Size distribution time histogram

MPINSIDE_CALLSTACK_DEPTH

Unwinds the stack up to the depth specified, and creates an mpinside_clstk.xxx file for each rank. The mpinside_clstk.xxx files can be post-processed with the utility **`MPInside_post -l`** to get the routine names and source line numbers. Use the **`-h`** option of **`MPInside_post`** to get more information.

MPINSIDE_EVAL_COLLECTIVE_WAIT

Checks whether a slow collective operation (such as MPI_Bcast or MPI_Allreduce) is caused by some ranks reaching the rendezvous point much later than others.

Setting this variable will put an MPI_Barrier and time it before any MPI collective operation. This assumes that the time of a collective operation is the sum of the time for all processors to synchronize plus the time of the actual operation (i.e, the physical transfer of data). In the mpinside_stats output file, the "b_*xxx*" column will give the MPI_Barrier time of the corresponding *xxx* MPI collective function, and the "*xxx*" column gives the remaining time.

MPINSIDE_EVAL_SLT

Checks whether a slow point-to-point communication is caused by late senders.

Setting this variable will measure the time the **`Send`**s arrived late compared to Recv or Wait arrivals. In the mpinside_stats output file, the column "w_*xxx*" (for example, w_wait or w_recv) represents the send-late-time (SLT) for the corresponding *xxx* function.

Note: The MPINSIDE_EVAL_COLLECTIVE_WAIT and MPINSIDE_EVAL_SLT variables are useful for identifying some problems that are intrinsic to an application, i.e., issues that cannot be remedied even with an ideal system (zero latency and infinite bandwidth).

References

- **mpinside** man page
- MPInside Reference Manual on the PFEs: /nasa/sgi/MPInside/3.6.2/doc/mpinside_3.5_ref_manual.pdf
- HPE Online Documentation: MPInside Reference Guide
- Publication: MPInside: a Performance Analysis and Diagnostic Tool for MPI Applications

Using the IOT Toolkit for I/O and MPI Performance Analysis

IOT is a licensed toolkit developed by I/O Doctors, LLC, for I/O and MPI instrumentation and optimization of high-performance computing programs. It allows flexible and user-controllable analysis at various levels of detail: per job, per MPI rank, per file, and per function call. In addition to information such as time spent, number of calls, and number of bytes transferred, IOT also provides the time of the I/O and/or MPI call, and where in the file the I/O occurs.

IOT can be used to analyze Fortran, C, and C++ programs as well as script-based applications, such as R, MATLAB, and Python. The toolkit works with multiple MPI implementations, including HPE MPT and Intel MPI. It is available for use on Pleiades, Electra, and Endeavour. Basic instructions are provided below. If you are interested in more advanced analysis, contact User Services at support@nas.nasa.gov.

Setting Up IOT

To set up IOT, complete the following steps. You only need to do these steps once.

1. Add **`/nasa/IOT/latest/bin64`** to your search path in your .cshrc file (for csh users) or .profile file (for bash users), as shown below. Be sure to add this line *above* the line in the file that checks for the existence of the prompt.

   For csh, use:
   `set path = ( $path /nasa/IOT/latest/bin64 )`

   For bash, use:
   `PATH=$PATH:/nasa/IOT/latest/bin64`
2. Run the **`iot -V`** command to check whether IOT is working. The output should be similar to the following example:

   ```
   % iot -V
   Using IOT install /nasa/IOT/v4.0.04/
           bin64/iot           v4.0.04 built Jun 14 2017 11:23:19
           lib64/libiot.so     v4.0.04 built Jun 14 2017 11:23:19
           libiotperm.so  v3.2.08 "Nasa_Advanced_SuperComputing" 5/11/2018 *
   ```
3. In your home directory, untar the **`/nasa/IOT/latest/ipsd/user_ipsd.tgz`** file:

   ```
   % tar xvzf /nasa/IOT/latest/ipsd/user_ipsd.tgz
   ```

   A directory called **`ipsd`** should be created under your $HOME directory.
4. Confirm that **`ipsd`** can be started:

   ```
   % ipsctl -A `hostname -s`
   No shares detected
   ```
5. Test IOT using the **`dd`** utility. First, create a directory called "dd" and change (**`cd`**) into it. Then, run the **`iot`** command as follows:

   ```
   % iot dd if=/dev/zero of=/dev/null count=20 bs=4096
   20+0 records in
   20+0 records out
   81920 bytes (82 kB) copied, 0.000123497 s, 663 MB/s
   ```

   In addition to the output shown above, you should also find a file called **`iot.`*`xxxxx`*`.ilz`**. The ILZ file is the output from the **`iot`** command.

Using IOT to Analyze Your Application

Once IOT is set up, follow these steps to analyze your application.

1. Create a configuration file that tells IOT what you want to instrument or monitor. You can use one of the following sample configuration files, which are available in the **`/nasa/IOT/latest/icf`** directory:

   trc_summary.icf
   : Use this file to start your first I/O analysis. This file provides a summary of information on the total counts, time spent, and bytes transferred for each I/O function of each file. For MPI applications, it also provides the same information obtained with mpi_summary.icf (described below).

   trc_interval.icf
   : In addition to the data collected by trc_summary.icf, this file provides more details for the read/write MPI function, per 1000-ms interval, including: the wall time when the calls occur; counts; time spent; and bytes transferred.

   trc_events.icf
   : In addition to the data collected by trc_summary.icf, this file provides the most details for each read/write function at the per-event level, including: the wall time when each call occurs; the time spent for the call; and the number of bytes transferred.

   mpi_summary.icf
   : Use this file to start your first MPI analysis. The file provides a summary of information such as the total count, time spent, and bytes transferred for all of the MPI functions called by the MPI ranks.

   mpi_interval.icf
   : In addition to the information collected by mpi_summary.icf, this file provides more details for the MPI functions, per 1000-millisecond (ms) interval, including: the wall time when the calls occur; counts; time spent; and bytes transferred.

   mpi_events.icf
   : This file provides the most details for each MPI function, at the per-event level, including: the wall time when each call occurs; the time spent for the call; and the number of bytes transferred.
2. Modify the **`mpiexec`** execution line in your PBS script to run IOT. For example, replace **`mpiexec -np 100 a.out`** with the following lines:

   ```
   set JOB_NUMBER=`echo $PBS_JOBID | awk -F. '{ print $1 }'`
    iot -m mpt -f cfg.icf -c '${ï»¿HOSTï»¿}':pfe22:`pwd`/a.out.collect.${ï»¿JOB_NUMBERï»¿}.ilz \
    mpiexec -np 100 a.out
   ```

   This method will generate an ILZ file named **`a.out.collect.$.ilz`**.

   Another option is to simply use:

   ```
   iot -m mpt -f cfg.icf \
    mpiexec -np 100 a.out
   ```

   This method will generate an ILZ file named **`iot.`*`process_id`*`.ilz`**.

   TIP: When the **`-m`** option is enabled, the default value for **`-f`** is **`mpi_summary.icf,`** which will be located automatically in the **`/nasa/IOT/latest/icf`** directory.

For more information, see **`iot -h`** on the **IOT Options** and **`iot -M`** on the **IOT Layers** man pages.

Viewing the ILZ File

Once your ILZ file is generated, you can view the data with the Pulse graphical user interface (GUI) using one of the following methods. Pulse will read in the data from the file and organize it for easy analysis in the GUI.

Run Pulse on Your Local System (Recommended)

Follow these steps to run Pulse on your local system.

1. Download Pulse.jar and the ILZ file:

   ```
   your_local_system% scp pfe:/nasa/IOT/latest/pulse.d/Pulse.jar .
   your_local_system% scp pfe:/path_to_ilz_file/filename.ilz .
   ```
2. Run Pulse through Java:

   ```
   your_local_system% java -jar Pulse.jar filename.ilz
   ```

Note: Download the latest version of Pulse.jar from time to time, as enhancements may be added.

Run Pulse from a PFE

Log into a PFE, load a Java module, and run Pulse:

```
pfe21% module load jvm/jrel.8.0_121
pfe21% pulse filename.ilz
```

TIPS:

- Pulse will uncompress the ILZ file to 4-5 times its compressed size. If the uncompressed file gets very large, Pulse may run out of memory. If this happens, you can try to increase memory using the **`java -Xmx4g`** option, as follows:

  ```
  % java -Xmx4g -jar Pulse.jar filename.ilz
  ```
- While Pulse is reading the file, the filename in the GUI will appear in red text. You can stop it before Pulse consumes too much memory by right-clicking the filename and selecting **Stop Reading**.

Additional Documentation

IOT documentation provided by the vendor is available in the **`/nasa/IOT/Doc`** directory.

Overview of Intel VTune Analysis

The Intel VTune Profiler is an analysis and tuning tool that provides predefined analysis configurations to address performance questions.

This article provides basic information on several Intel VTune Profiler analysis types that examine various aspects of performance and identify potential benefits for your application from available hardware resources.

VTune Profiler Analysis Types

The following list provides a brief description of each analysis type that can be used with NAS resources.

performance-snapshot
: Get a quick snapshot of your application performance and identify next steps for deeper analysis. This analysis type became available starting with VTune Profiler Version 2020 Update 2. The Intel Xeon Sandy Bridge processors are not supported for this analysis.

hotspots
: Investigate call paths and find where your code is spending the most time; identify opportunities to tune your algorithms. This analysis type is in the Vtune Profiler's Algorithm analysis group.

threading
: Discover how well your application is using parallelism to take advantage of all available CPU cores. Best for visualizing thread parallelism on available cores, locating causes of low concurrency, and identifying serial bottlenecks in your code. This analysis type is in the Parallelism analysis group.

memory-consumption
: Analyze memory consumption by your application, its distinct memory objects, and their allocation stacks. This analysis type is in the Algorithm analysis group.

uarch-exploration
: Analyze CPU microarchitecture bottlenecks affecting the performance of your application. This analysis type is in the Microarchitecture analysis group.

memory-access
: Measure a set of metrics to identify issues related to memory access. Best for memory-bound applications to determine which level of the memory hierarchy is impacting your performance by reviewing CPU cache and main memory usage, including possible NUMA issues. This analysis type is in the Microarchitecture analysis group.

hpc-performance
: Analyze performance aspects of compute-intensive applications, including CPU and GPU utilization. Get information on OpenMP efficiency, memory access, and vectorization. This analysis type is in the Parallelism analysis group.

io
: Analyze utilization of IO subsystems, CPUs, and processor buses. This analysis type is in the Input and Output analysis group.

VTune Profiler Collection Types

There are two collection types:

- User-mode sampling and tracing collection.
- Hardware event-based sampling collection.

Each VTune analysis type uses one of the two collection types by default.

User-Mode Sampling and Tracing Collection

This collection method uses the operating system interrupts. No sampling drivers are needed. Default resolution is 10 millisecond (ms). The method works for the Intel Xeon nodesâ both Pleiades front ends (PFEs) and computeâ and the AMD Rome nodes.

Hardware Event-based Sampling (EBS) Collection

This collection method uses the counter overflow feature of the on-chip Performance Monitoring Unit (PMU). Default resolution is 1 ms. The is enabled for the Intel Xeon compute nodes only; it is not applicable to the Pleiades front-end nodes (for security reasons) or the AMD Rome processors (for hardware reasons).

There are two ways to facilitate EBS:

- Install and load the Intel sampling drivers
- Enable the Perf driverless collection, where the Linux Perf driver is used instead of the Intel sampling drivers

Each method is described below. Both require administrator privilege to configure them.

Install and Load the Intel Sampling Drivers

Different VTune Profiler versions may need different versions of the sampling drivers. A change in the kernel may result in the need to reinstall and reload the drivers. Here is one way to check whether the VTune sampling drivers have been loaded on the system for a particular version of VTune:

```
% <VTUNE_PROFILER_DIR>/sepdk/src insmod-sep -q
```

For example, running the command as shown below on a compute node will show that the drivers have been loaded:

```
% /nasa/intel/Compiler/2021.4.0/vtune/2021.9.0/sepdk/src insmod-sep -q
```

Enable the Perf Driverless Collection

With this method, the Linux Perf driver is used instead of the Intel sampling drivers. The Intel VTune Profiler can use this driverless mode if the following requirements are satisfied:

- Access to core and uncore events. All hardware event-based collections in VTune Profiler use core PMU events. Some of them, such as the Memory Access and IO analysis types, also require access to uncore events that enable collecting metrics such as DRAM bandwidth, QPI/UPI bandwidth, PCI bandwidth, and others.
- Perf for Linux kernel 2.6.32 or later. PMU events are exposed by the Linux kernel through **`/sys/bus/event_source/devices/cpu`** and **`/sys/bus/event_source/devices/uncore_*`** directories.
- The value of **`/proc/sys/kernel/perf_event_paranid`** must be 0 or 1.

See Intel Processor Events Reference to learn more about core and uncore events.

Compared to the Intel sampling drivers method, there are limitations to using the Perf driverless collection mode. For example, in order to use Perf driverless collection mode with the uarch-exploration and memory-access analysis types, a system administrator must set the default limit of opened file descriptors (listed in the **`/etc/security/limits.conf`** file) to exceed 100 x ***`number_of_logic_CPU_cores`***. More limitations are listed in the Intel documentation, Profiling Hardware Without Intel Sampling Drivers.

Note: On HECC Intel Xeon compute nodes running the TOSS3 image, both EBS collection modes are enabled for vtune/2021.9.

In all cases except for using the hotspots analysis type with stack collection, VTune Profiler uses the Intel sampling drivers if they are loaded. To make the VTune Profiler use the driverless Perf mode for sampling without stacks, create a custom analysis type and select the **Enable driverless collection** option in the GUI, or set the knob value in the command line to enable-driverless-collection=true as follows:

```
vtune -collect-with runsa -knob enable-driverless-collection=true \
-knob event-config=<event-list> <application>
```

For example,

```
-knob enable-driverless-collection=true -knob \
event-config=CPU_CLK_UNHALTED.CORE,CPU_CLK_UNHALTED.REF,INST_RETIRED.ANY
```

The **`enable-driverless-collection`** option is available starting with VTune 2019 Update 4.

Note: The command lines above is too long to be formatted as one line, so they are broken with a backslash (\).

Readiness of Running Various Analysis Types

To check whether VTune Profiler is ready for use on a system, you can use the **`vtune-self-checker.sh`** script, which can be found under <VTUNE_PROFILER_DIR>/bin64. This script runs a subset of the available analysis types against a matrix multiply application, and reports whether the analysis runs successfully. If EBS is used for the analysis, it also states whether the Intel sampling drivers or the Perf driverless mode is used.

Using the VTune Profiler Command Line

We recommend using the latest version of VTune on HECC systems, vtune/2021.9:

```
module load vtune/2021.9
```

To learn more about various analysis types, use:

```
vtune -h collect
```

```
or
```

```
vtune -h collect name_of_analysis_type
```

To run the VTune self checker (to confirm whether the correct drivers are installed and the system is set up properly):

```
vtune-self-checker.sh > output
```

When using the HPE MPT libraries with VTune, set the MPI_SHEPHERD and MPI_USING_VTUNE variables as follows.

Note: For some analyses, MPT may generate error message in your PBS output files stating that additional variables (such as MPI_UNBUFFERED_STDIO) need to be set.

```
(bash)
export MPI_SHEPHERD=true
export MPI_USING_VTUNE=true

(csh)
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
```

For an example of running an analysis type and viewing the results, see Finding Hotspots in Your Code with the Intel VTune Command-Line Interface.

Summary Table

The information provided above was tested on NAS systems using **`vtune/2021.9`** and is summarized in the following table.

Analysis Type	Additional Knobs Used	Analysis Group	Sampling Method	If EBS, Which Mode?	Unsu Pro T
performance-snapshot		N/A	EBS	Intel Driver	Sand Rome
hotspots		Algorithm	User-Mode	N/A	None
hotspots	-knob sampling-mode=hw	Algorithm	EBS	Intel Driver	Rome
hotspots	-knob sampling-mode=hw -knob enable-stack-collection=true	Algorithm	EBS	Perf[1]	Rome
threading		Parallelism	User-Mode	N/A	None
threading	-knob sampling-and-waits=hw	Parallelism	EBS	Intel Driver	Rome
threading	-knob sampling-and-waits=hw -knob enable-stack-collection=true	Parallelism	EBS	Intel Driver	Rome
memory-consumption		Algorithm	User-Mode	N/A	None
uarch-exploration		Microarchitecture	EBS	Intel Driver	Hasw Rome
memory-access		Microarchitecture	EBS	Intel Driver	Rome
hpc-performance		Parallelism	EBS	Intel Driver	Sand Bridg
io		Input and Output	EBS	Intel Driver	Rome
-collect-with runsa	-knob event-config=<event-list>	Custom Analysis	EBS	Intel Driver	Rome

-collect-with runsa	-knob event-config=<event-list> -knob enable-driverless-collection=true	Custom Analysis	EBS	Perf	Rome

[1] Starting with VTune 2019 Update 4, hotspots with EBS and stacks uses the Perf driverless mode by default even when the Intel sampling drivers are available.

[2] No L2, L3, DRAM bound (in % of clockticks), memory bandwidth or memory latency available for Haswell when hyperthreading is turned on.

[3] Vectorization analysis is limited for Sandy Bridge. Only metrics based on binary static analysis such as vector instruction set will be available.

Additional References

- VTune Analysis Types
- Intel VTune Profiler User Guide
- Intel VTune Profiler Performance Analysis Cookbook

Finding Hotspots in Your Code with the Intel VTune Command-Line Interface

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Pleiades, Aitken, and Electra Intel processor types: Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the **`-g`** option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, such as **`mpi-hpe/mpt.2.25`**, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

   ```
   module load vtune/2021.9
   ```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the **`vtune`** command line that is appropriate for your code, as listed below. Use the **`-collect`** (or **`-c`**) option to run the hotspots collection and the **`-result-dir`** (or **`-r`**) option to specify a directory.

Note: The **`vtune`** command replaces **`amplxe-cl`**, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs
```

Data collected by VTune for the **`a.out`** application are stored in the **`r000hs`** directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in the **r000hs** directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir
```

Note: If your **vtune** run fails with **MPT ERROR** and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default to **stdout**. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, **r000hs.r587i0n0** and **r000hs.r587i0n1**, where each directory contains 24 subdirectories, (**data.[*0-23*]**).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hsÂ  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), **vtune-gui**. Since the Pleiades front-end systems (PFEs) do not have sufficient memory to run **vtune-gui**, run it on a compute node as follows:

1. On a PFE, run **echo $DISPLAY** to find its setting (for example, **pfe22:102.0**).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session
   (**qsub -I -X**) or a reservable front end (**pbs_rfe**).
5. On the compute node, start **vtune-gui** with name of the result directory, for example, **r000hs**.

   ```
   compute_node% module load vtune/2021.9
   compute_node% vtune-gui r000hs
   ```

Generating a Report

You can also use the **vtune** command with the **-report** option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to **stdout**, or to a specific output text file, such as **output.txt:**

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```
- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```
- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```
- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```
- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Using Darshan for I/O Characterization

Darshan is an open-source, lightweight high-performance computing I/O characterization toolkit developed at Argonne National Lab. Darshan instruments applications and captures calls to the POSIX, MPI-IO, STDIO, and HDF5 interfaces, and provides some limited information about PnetCDF. Statistics collected by Darshan include (but are not limited to) the number, size, and time spent on I/O calls, as well as the number and names of files, POSIX I/O access patterns (such as the access sizes and the numbers of total, consecutive, and sequential operations), timespan from first to last access, and estimated performance.

Darshan on Pleiades

Two different Darshan builds are available. One was built on a Pleiades front end (PFE) with the TOSS 3 operating system for MPI applications that use HPE MPT, and the other was built for non-MPI applications. Both are described below. Note that the two builds are installed in different locations.

Darshan for HPE MPI Applications

After loading an Intel compiler module and an HPE MPT module, this Darshan runtime was built with these options:

```
--prefix=/nasa/darshan/3.4.0_mpt
--with-log-path-by-env=DARSHAN_LOG_PATH
--enable-hdf5-mod
--with-hdf5=/nasa/hdf5/1.12.0_mpt
--with-jobid-env=PBS_JOBID CC=mpicc
```

Darshan for Non-MPI Applications

This Darshan runtime was built with these options:

```
--prefix=/nasa/darshan/3.4.0_non_mpi --with-log-path-by-env=DARSHAN_LOG_PATH
--with-jobid-env=PBS_JOBID --without-mpi CC=gcc
```

Instrumenting Your Application Using darshan-runtime

You can enable Darshan instrumentation either at compile/link time or at runtime. Runtime instrumentation is a quicker way to get started using Darshan, as it does not require modification to your existing executable; rather, it's done by setting the LD_PRELOAD environment variable for your PBS batch job as described below.

As the application runs, Darshan records statistics for each process. At the end of the run, Darshan collects, aggregates, compresses, and writes a log file to a location specified by the environment variable DARSHAN_LOG_PATH. The filename of the Darshan binary log will likely follow this format:

`<USERNAME>_<BINARY_NAME>_<JOB_ID>_<DATE>_<UNIQUE_ID>_<TIMING>.darshan.`

You can change the location and name of the Darshan log by setting the variable DARSHAN_LOGFILE=/path/*name*.darshan.

Using darshan-runtime for HPE MPI Applications

Use the following PBS script to run your HPE MPI application with Darshan.

Note: For MPI applications, the MPI_SHEPHERD environment variable must be set to true (highlighted in the script below) to avoid deadlock when preloading the Darshan shared library.

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
module load mpi-hpe/mpt.2.25
#Note: Profiling HDF5 app with Darshan should NOT use hdf5 version 1.8.18_mpt
#Uncomment the following two lines if running HDF5 applications
#module use -a /nasa/modulefiles/testing
#module load hdf5/1.12.0_mpt

export DARSHAN_DIR=/nasa/darshan/3.4.0_mpt
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export MPI_SHEPHERD=true

mpiexec -genv LD_PRELOAD ${DARSHAN_DIR}/lib/libdarshan.so -np X mpi_executable
```

TIP: By default, Darshan aggregates statistics for files accessed by all ranks and collapses them into a single cumulative file record. If you want to retain the per-process information (which will result in a larger log file), set this variable:

`export DARSHAN_DISABLE_SHARED_REDUCTION=1`

Using darshan-runtime for Non-MPI Applications

Use the following PBS script to run your non-MPI application with Darshan.

Note: For non-MPI applications, the DARSHAN_ENABLE_NONMPI environment variable must be set to 1 (highlighted in the script, below).

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
export DARSHAN_DIR=/nasa/darshan/3.4.0_non_mpi
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export DARSHAN_ENABLE_NONMPI=1
env LD_PRELOAD=${DARSHAN_DIR}/lib/libdarshan.so non-mpi_executable
```

Note: Darshan does not necessarily interfere with other profiling tools, such as the NAS-developed **`mpiprof`** tool. However, outputs generated by other profiling tools will likely also be counted in the Darshan binary log, which may confuse your analysis.

Darshan Extended Tracing

Darshan can perform eXtended Tracing (DXT) of the MPI-IO and POSIX I/O to provide details on each read or write operation issued by each rank, as well as which ranks are performing I/O and how long they are spending on I/O. A memory limit of 2 MebiByte (MiB) each for DXT_MPIIO and DXT_POSIX modules is set as default.

Note: DXT may result in longer runtime and higher memory overheads. It is also possible that the tracing log may be incomplete if a Darshan DXT module runs out of memory to store new record data.

To use DXT to trace all files, add one of the following lines in your PBS script:

```
export DARSHAN_MOD_ENABLE="DXT_POSIX,DXT_MPIIO"
or
export DXT_ENABLE_IO_TRACE=1
(This is a variable for earlier Darshan versions. It also works with 3.4.0.)
```

You can also select which items you want to include in or exclude from DXT (for exampleâ   files, ranks, or small I/O operations) by providing a trace configuration file or by setting certain DXT environment variables. For more information, read Sections 6 and 8 of the Darshan-runtime installation and usage documentation.

Analyzing Darshan Log Files with darshan-util Tools

The Darshan log file generated can be ported to and analyzed on systems where **`darshan-util`** is installed. You can perform the analysis on a PFE by loading either the HPE MPI version or the non-MPI version of the Darshan modulefiles, as shown below.

Note: Some of the **`darshan-util`** tools also require **`Perl`**, **`pdflatex`**, **`gnuplot`**, and **`epstopdf`**, which can be found in the pkgsrc/2021Q2 directory on Pleiades.

```
pfe% module use -a /nasa/modulefiles/testing

pfe% module load darshan/3.4.0_mpt
or
pfe% module load darshan/3.4.0_non_mpi

pfe% module load pkgsrc/2021Q2
```

After you load the Darshan and pkgsrc modulefiles, you can use one of the following **`darshan-util`** tools on a PFE.

darshan-job-summary.pl

The **`darshan-job-summary.pl`** tool processes the Darshan binary log (for example, ***`name`*`.darshan`**) and generates a multi-page PDF (***`name`*`.darshan.pdf`**) containing graphs, tables, and performance estimates characterizing the I/O activity of the job:

```
pfe% darshan-job-summary.pl name.darshan
```

A sample PDF provided by the Darshan developers can be found here.

darshan-summary-per-file.sh

This script is similar to **`darshan-job-summary.pl`** except that it produces a separate PDF summary for every file accessed by the application:

```
pfe% darshan-summary-per-file.sh name.darshan output-dir
```

where ***`output-dir`*** is a directory to be created; it will contain the collection of PDFs (one PDF per file).

Note: Using this utility is not recommended if your application opens a large number of files.

darshan-parser

The `darshan-parser` tool extracts everything from a Darshan log and displays it in text format. It provides more information than `darshan-job-summary.pl`.

```
pfe% darshan-parser [options] name.darshan > name.txt
```

To show the available options, do: `darshan-parser -h`

To extract information for a specific file from a Darshan log containing statistics for many files, and produce a Darshan log for that file, complete these steps:

1. Show a list of filenames along with each file's record ID:
   `darshan-parser --file-list name.darshan`
2. Generate the log for the file you want:
   `darshan-convert --file a_record_id name.darshan a_file.darshan`.

darshan-dxt-parser

If you used DXT to generate a trace file (for example, `dxt_name.darshan`), use the `darshan-dxt-parser` tool to generate a text output:

```
pfe% darshan-dxt-parser [--show-incomplete] dxt_name.darshan > dxt_name.txt
```

The option `--show-incomplete` will display results even if the log is incomplete.

Known Issues

When you post-process the Darshan binary log for runs performed under a Lustre filesystem with progressive file layout, the following error will occur:

```
Error: failed to parse LUSTRE module record.
```

However, this error does not affect other I/O statistics (such as the POSIX or MPIIO stats) collected by Darshan for your application.

References

- Darshan-runtime installation and usage (version 3.4.0)
- Darshan-util installation and usage (version 3.4.0)

Process/Thread Pinning

Instrumenting Your Fortran Code to Check Process/Thread Placement

Pinning, the binding of a process or thread to a specific core, can improve the performance of your code.

You can insert the MPI function **`mpi_get_processor_name`** and the Linux C function **`sched_getcpu`** into your source code to check process and/or thread placement. The MPI function **`mpi_get_processor_name`** returns the hostname an MPI process is running on (to be used for MPI and/or MPI+OpenMP codes only). The Linux C function **`sched_getcpu`** returns the processor number the process/thread is running on.

If your source code is written in Fortran, you can use the C code **`mycpu.c`**, which allows your Fortran code to call **`sched_getcpu`**. The next section describes how to use the **`mycpu.c`** code.

C Program mycpu.c

```
#include <utmpx.h>
int sched_getcpu();

int findmycpu_ ()
{
    int cpu;
    cpu = sched_getcpu();
    return cpu;
}
```

Compile **`mycpu.c`** as follows to produce the object file **`mycpu.o`**:

```
pfe21% module load comp-intel/2020.4.304
pfe21% icc -c mycpu.c
```

The following example demonstrates how to instrument an MPI+OpenMP source code with the above functions. The added lines are highlighted.

```
      program your_program
      use omp_lib
...
      integer :: resultlen, tn, cpu
      integer, external :: findmycpu
      character (len=8) :: name

      call mpi_init( ierr )
      call mpi_comm_rank( mpi_comm_world, rank, ierr )
      call mpi_comm_size( mpi_comm_world, numprocs, ierr )
      call mpi_get_processor_name(name, resultlen, ierr)
!$omp parallel

      tn = omp_get_thread_num()
      cpu = findmycpu()
      write (6,*) 'rank ', rank, ' thread ', tn,
     &  ' hostname ', name, ' cpu ', cpu
.....
!$omp end parallel
      call mpi_finalize(ierr)
      end
```

Compile your instrumented code as follows:

```
pfe21% module load comp-intel/2020.4.304
pfe21% module load mpi-hpe/mpt
pfe21% ifort -o a.out -qopenmp mycpu.o your_program.f -lmpi
```

Sample PBS script

The following PBS script provides an example of running the hybrid MPI+OPenMP code across two nodes, with 2 MPI processes per node and 4 OpenMP threads per process, and using the mbind tool to pin the processes and threads.

```
#PBS -lselect=2:ncpus=28:mpiprocs=2:model=bro
#PBS -lwalltime=0:10:00

cd $PBS_O_WORKDIR

module load comp-intel/2020.4.304
module load mpi-hpe/mpt

mpiexec -np 4 mbind.x -cs -t4 -v ./a.out
```

Here is a sample output:

```
These 4 lines are generated by mbind only if you have included the -v option:
host: r627i4n1, ncpus: 56, rank: 0 (r0), nthreads: 4, bound to cpus: {0-9:3}
host: r627i4n1, ncpus: 56, rank: 1 (r1), nthreads: 4, bound to cpus: {14-23:3}
host: r627i4n8, ncpus: 56, rank: 2 (r0), nthreads: 4, bound to cpus: {0-9:3}
host: r627i4n8, ncpus: 56, rank: 3 (r1), nthreads: 4, bound to cpus: {14-23:3}

These lines are generated by your instrumented code:
rank    0 thread    3 hostname r627i4n1 cpu    0
rank    0 thread    3 hostname r627i4n1 cpu    3
rank    0 thread    3 hostname r627i4n1 cpu    6
rank    0 thread    3 hostname r627i4n1 cpu    9
rank    1 thread    3 hostname r627i4n1 cpu   14
rank    1 thread    3 hostname r627i4n1 cpu   17
rank    1 thread    3 hostname r627i4n1 cpu   20
rank    1 thread    3 hostname r627i4n1 cpu   23
rank    2 thread    3 hostname r627i4n8 cpu    0
rank    2 thread    3 hostname r627i4n8 cpu    3
rank    2 thread    3 hostname r627i4n8 cpu    6
rank    2 thread    3 hostname r627i4n8 cpu    9
rank    3 thread    3 hostname r627i4n8 cpu   14
rank    3 thread    3 hostname r627i4n8 cpu   17
rank    3 thread    3 hostname r627i4n8 cpu   20
rank    3 thread    3 hostname r627i4n8 cpu   23
```

Note: In your output, these lines may be listed in a different order.

Using Intel OpenMP Thread Affinity for Pinning

UPDATE IN PROGRESS: This article is being updated to support Skylake and Cascade Lake.
The Intel compiler's OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. Depending on the system topology, application, and operating system, thread affinity can have a dramatic effect on code performance. We recommend two approaches for using the Intel OpenMP thread affinity capability.

Using the KMP_AFFINITY Environment Variable

The thread affinity interface is controlled using the KMP_AFFINITY environment variable.

Syntax

For **`csh`** and **`tcsh`**:

```
setenv KMP_AFFINITY [<modifier>,...]<type>[,<permute>][,<offset>]
```

For **`sh, bash,`** and **`ksh`**:

```
export KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

Using the Compiler Flag -par-affinity Compiler Option

Starting with the Intel compiler version 11.1, thread affinity can be specified through the compiler option **`-par-affinity`**. The use of **`-openmp`** or **`-parallel`** is required in order for this option to take effect. This option overrides the environment variable when both are specified. See **man ifort** for more information.

Note: Starting with **`comp-intel/2015.0.090`**, **`-openmp`** is deprecated and has been replaced with **`-qopenmp`**.

Syntax

```
-par-affinity=[<modifier>,...]<type>[,<permute>][,<offset>]
```

Possible Values for type

For both of the recommended approaches, the only required argument is **`type`**, which indicates the type of thread affinity to use. Descriptions of all of the possible arguments (**`type`**, **`modifier`**, **`permute`**, and **`offset`**) can be found in **`man ifort`**.

Recommendation: Use Intel compiler versions 11.1 and later, as some of the affinity methods described below are not supported in earlier versions.

Possible values for **`type`** are:

type = none (default)
Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to

determine machine topology. Specify **`KMP_AFFINITY=verbose,none`** to list a machine topology map.

type = disabled

Specifying **`disabled`** completely disables the thread affinity interfaces. This forces the OpenMP runtime library to behave as if the affinity interface was not supported by the operating system. This includes implementations of the low-level API interfaces such as **`kmp_set_affinity`** and **`kmp_get_affinity`** that have no effect and will return a nonzero error code.

Additional information from Intel:

"The thread affinity type of KMP_AFFINITY environment variable defaults to none (**`KMP_AFFINITY=none`**). The behavior for **`KMP_AFFINITY=none`** was changed in 10.1.015 or later, and in all 11.*x* compilers, such that the initialization thread creates a "full mask" of all the threads on the machine, and every thread binds to this mask at startup time. It was subsequently found that this change may interfere with other platform affinity mechanism, for example, **`dplace()`** on Altix machines. To resolve this issue, a new affinity type **`disabled`** was introduced in compiler 10.1.018, and in all 11.*x* compilers (**`KMP_AFFINITY=disabled`**). Setting **`KMP_AFFINITY=disabled`** will prevent the runtime library from making any affinity-related system calls."

type = compact

Specifying **`compact`** causes the threads to be placed as close together as possible. For example, in a topology map, the nearer a core is to the root, the more significance the core has when sorting the threads.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=compact,verbose

# for csh, tcsh
setenv KMP_AFFINITY compact,verbose
```

type = scatter

Specifying **`scatter`** distributes the threads as evenly as possible across the entire system. Scatter is the opposite of compact.

Note: For most OpenMP codes, **`type=scatter`** should provide the best performance, as it minimizes cache and memory bandwidth contention for all processor models.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=scatter,verbose

# for csh, tcsh
setenv KMP_AFFINITY scatter,verbose
```

type = explicit

Specifying **`explicit`** assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the **`proclist=`*`modifier`***, which is required for this affinity type.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY="explicit,proclist=[0,1,4,5],verbose"
```

```
# for csh, tcsh
setenv KMP_AFFINITY "explicit,proclist=[0,1,4,5],verbose"
```

For nodes that support hyperthreading, you can use the **`granularity`** modifier to specify whether to pin OpenMP threads to physical cores using **`granularity=core`** (the default) or pin to logical cores using **`granularity=fine`** or **`granularity=thread`** for the **`compact`** and **`scatter`** types.

Examples

The following examples illustrate the thread placement of an OpenMP job with four threads on various platforms with different thread affinity methods. The variable OMP_NUM_THREADS is set to 4:

```
# for sh, ksh, bash
export OMP_NUM_THREADS=4

# for csh, tcsh
setenv OMP_NUM_THREADS 4
```

The use of the **`verbose`** modifier is recommended, as it provides an output with the placement.

Sandy Bridge (Pleiades)

As seen in the configuration diagram of a Sandy Bridge node, each set of eight physical cores in a socket share the same L3 cache.

Four threads running on 1 node (16 physical cores and 32 logical cores due to hyperthreading) of Sandy Bridge will get the following thread placement:

kb285_sandybridge_table.png

Ivy Bridge (Pleiades)

As seen in the configuration diagram of an Ivy Bridge node, each set of ten physical cores in a socket share the same L3 cache.

Four threads running on 1 node (20 physical cores and 40 logical cores due to hyperthreading) of Ivy Bridge will get the following thread placement:

IvyBridgeTableThumb.png

Haswell (Pleiades)

As seen in the configuration diagram of a Haswell node, each set of 12 physical cores in a socket share the same L3 cache.

Four threads running on 1 node (24 physical cores and 48 logical cores due to hyperthreading) of Haswell will get the following thread placement:

HaswellTableThumb_1.png

Broadwell (Pleiades and Electra)

As seen in the configuration diagram of a Broadwell node, each set of 14 physical cores in a socket share the same L3 cache.

Four threads running on 1 node (28 physical cores and 56 logical cores due to hyperthreading) of Broadwell will get the following thread placement:

BroadwellTableThumb_1_1.png

Using HPE MPT Environment Variables for Pinning

For MPI codes built with HPE's MPT libraries, one way to control pinning is to set certain MPT memory placement environment variables. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

MPT Environment Variables

Here are the MPT memory placement environment variables:

MPI_DSM_VERBOSE

Directs MPI to display a synopsis of the NUMA and host placement options being used at run time to the standard error file.

Default: Not enabled

The setting of this environment variable is ignored if MPI_DSM_OFF is also set.

MPI_DSM_DISTRIBUTE

Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. Currently, the CPUs are chosen by simply starting at relative CPU 0 and incrementing until all MPI processes have been forked.

Default: Enabled

WARNING: If the nodes used by your job are not fully populated with MPI processes, use **`MPI_DSM_CPULIST`**, **`dplace`**, or **`omplace`** for pinning instead of **`MPI_DSM_DISTRIBUTE`**.
The **`MPI_DSM_DISTRIBUTE`** setting is ignored if **`MPI_DSM_CPULIST`** is also set, or if **`dplace`** or **`omplace`** are used.

MPI_DSM_CPULIST

Specifies a list of CPUs on which to run an MPI application, excluding the shepherd process(es) and **`mpirun`**. The number of CPUs specified should equal the number of MPI processes (excluding the shepherd process) that will be used.

Syntax and examples for the list:

- Use a comma and/or hyphen to provide a delineated list:

```
# place MPI processes ranks 0-2 on CPUs 2-4
# and ranks 3-5 on CPUs 6-8
setenv MPI_DSM_CPULIST "2-4,6-8"
```

- Use a "/" and a stride length to specify CPU striding:

```
# Place the MPI ranks 0 through 3 stridden
# on CPUs 8, 10, 12, and 14
setenv MPI_DSM_CPULIST 8-15/2
```

- Use a colon to separate CPU lists of multiple hosts:

```
# Place the MPI processes 0 through 7 on the first host
# on CPUs 8 through 15. Place MPI processes 8 through 15
# on CPUs 16 to 23 on the second host.
setenv MPI_DSM_CPULIST 8-15:16-23
```

- Use a colon followed by **`allhosts`** to indicate that the prior list pattern applies to all subsequent hosts/executables:

```
# Place the MPI processes onto CPUs 0, 2, 4, 6 on all hosts
setenv MPI_DSM_CPULIST 0-7/2:allhosts
```

Examples

An MPI job requesting 2 nodes on Pleiades and running 4 MPI processes per node will get the following placements, depending on the environment variables set:

```
#PBS -lselect=2:ncpus=8:mpiprocs=4
module load <mpt_module>
setenv ....
cd $PBS_O_WORKDIR
mpiexec -np 8 ./a.out
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_DISTRIBUTE

```
MPI: DSM information
MPI: MPI_DSM_DISTRIBUTE enabled
grank   lrank   pinning  node name      cpuid
    0       0   yes      r86i3n5            0
    1       1   yes      r86i3n5            1
    2       2   yes      r86i3n5            2
    3       3   yes      r86i3n5            3
    4       0   yes      r86i3n6            0
    5       1   yes      r86i3n6            1
    6       2   yes      r86i3n6            2
    7       3   yes      r86i3n6            3
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6

```
MPI: WARNING MPI_DSM_CPULIST CPU placement spec list is too short.
MPI:         MPI processes on host #1 and later will not be pinned.
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r22i1n7            0
    1       1   yes      r22i1n7            2
    2       2   yes      r22i1n7            4
    3       3   yes      r22i1n7            6
    4       0   no      r22i1n8             0
    5       1   no      r22i1n8             0
    6       2   no      r22i1n8             0
    7       3   no      r22i1n8             0
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:0,2,4,6

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r13i2n12           0
    1       1   yes      r13i2n12           2
    2       2   yes      r13i2n12           4
    3       3   yes      r13i2n12           6
    4       0   yes      r13i3n7            0
    5       1   yes      r13i3n7            2
    6       2   yes      r13i3n7            4
    7       3   yes      r13i3n7           6
```

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:allhosts

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r13i2n12          0
    1       1   yes      r13i2n12          2
    2       2   yes      r13i2n12          4
    3       3   yes      r13i2n12          6
    4       0   yes      r13i3n7           0
    5       1   yes      r13i3n7           2
    6       2   yes      r13i3n7           4
    7       3   yes      r13i3n7           6
```

Using the omplace Tool for Pinning

HPE's **`omplace`** is a wrapper script for **`dplace`**. It pins processes and threads for better performance and provides an easier syntax than **`dplace`** for pinning processes and threads.

The **`omplace`** wrapper works with HPE MPT as well as with Intel MPI. In addition to pinning pure MPI or pure OpenMP applications, **`omplace`** can also be used for pinning hybrid MPI/OpenMP applications.

A few issues with **`omplace`** to keep in mind:

- **`dplace`** and **`omplace`** do not work with Intel compiler versions 10.1.015 and 10.1.017. Use the Intel compiler version 11.1 or later, instead
- To avoid interference between **`dplace/omplace`** and Intel's thread affinity interface, set the environment variable KMP_AFFINITY to disabled or set OMPLACE_AFFINITY_COMPAT to ON
- The **`omplace`** script is part of HPE's MPT, and is located under the /nasa/hpe/mpt/*mpt_version_number*/bin directory

Syntax

```
For OpenMP:
setenv OMP_NUM_THREADS nthreads
omplace [OPTIONS] program args...
or
omplace -nt nthreads [OPTIONS] program args...

For MPI:
mpiexec -np nranks omplace [OPTIONS] program args...

For MPI/OpenMP hybrid:
setenv OMP_NUM_THREADS nthreads
mpiexec -np nranks omplace [OPTIONS] program args...
or
mpiexec -np nranks omplace -nt nthreads [OPTIONS] program args...
```

Some useful **`omplace`** options are listed below:

WARNING: For **`omplace`**, a blank space is required between **`-c`** and **`cpulist`**. Without the space, the job will fail. This is different from **`dplace`**.

`-b` ***`basecpu`***
: Specifies the starting CPU number for the effective CPU list.

`-c` ***`cpulist`***
: Specifies the effective CPU list. This is a comma-separated list of CPUs or CPU ranges.

`-nt` ***`nthreads`***
: Specifies the number of threads per MPI process. If this option is unspecified, it defaults to the value set for the OMP_NUM_THREADS environment variable. If OMP_NUM_THREADS is not set, then *nthreads* defaults to 1.

`-v`
: Verbose option. Portions of the automatically generated placement file will be displayed.

`-vv`
: Very verbose option. The automatically generated placement file will be displayed in its entirety.

For information about additional options, see **man omplace**.

Examples

For Pure OpenMP Codes Using the Intel OpenMP Library

Sample PBS script:

```
#PBS -lselect=1:ncpus=12:model=wes

module load comp-intel/2015.0.090
setenv KMP_AFFINITY disabled

omplace -c 0,3,6,9 -vv ./a.out
```

Sample placement information for this script is given in the application's stout file:

```
omplace: placement file /tmp/omplace.file.21891
    firsttask cpu=0
    thread oncpu=0 cpu=3-9:3 noplace=1  exact
```

The above placement output may not be easy to understand. A better way to check the placement is to run the **ps** command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample output of placement.out

```
PSR COMMAND             TIME   PID  PPID   LWP
  0 openmp1         00:00:02 31918 31855 31918
 19 openmp1         00:00:00 31918 31855 31919
  3 openmp1         00:00:02 31918 31855 31920
  6 openmp1         00:00:02 31918 31855 31921
  9 openmp1         00:00:02 31918 31855 31922
```

Note that Intel OpenMP jobs use an extra thread that is unknown to the user, and does not need to be placed. In the above example, this extra thread is running on logical core number 19.

For Pure MPI Codes Using HPE MPT

Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/2015.0.090
module load mpi-hpe/mpt

#Setting MPI_DSM_VERBOSE allows the placement information
#to be printed to the PBS stderr file

setenv MPI_DSM_VERBOSE

mpiexec -np 8 omplace -c 0,3,6,9 ./a.out
```

Sample placement information for this script is shown in the PBS stderr file:

```
MPI: DSM information
MPI: using dplace
grank   lrank   pinning  node name      cpuid
    0       0   yes      r144i3n12         0
    1       1   yes      r144i3n12         3
    2       2   yes      r144i3n12         6
    3       3   yes      r144i3n12         9
    4       0   yes      r145i2n3          0
    5       1   yes      r145i2n3          3
    6       2   yes      r145i2n3          6
    7       3   yes      r145i2n3          9
```

In this example, the four processes on each node are evenly distributed to the two sockets (CPUs 0 and 3 are on the first socket while CPUs 6 and 9 on the second socket) to minimize contention. If **omplace** had not been used, then placement would follow the rules of the environment variable OMP_DSM_DISTRIBUTE, and all four processes would have been placed on the first socket -- likely leading to more contention.

For MPI/OpenMP Hybrid Codes Using HPE MPT and Intel OpenMP

Proper placement is more critical for MPI/OpenMP hybrid codes than for pure MPI or pure OpenMP codes. The following example demonstrates the situation when no placement instruction is provided and the OpenMP threads for each MPI process are stepping on one another which likely would lead to very bad performance.

Sample PBS script without pinning:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/2015.0.090
module load mpi-hpe/mpt
setenv OMP_NUM_THREADS 2

mpiexec -np 8 ./a.out
```

There are two problems with the resulting placement shown in the example above. First, you can see that the first four MPI processes on each node are placed on four cores (0,1,2,3) of the same socket, which will likely lead to more contention compared to when they are distributed between the two sockets.

```
MPI: MPI_DSM_DISTRIBUTE enabled
grank   lrank   pinning  node name      cpuid
    0       0   yes      r212i0n10          0
    1       1   yes      r212i0n10          1
    2       2   yes      r212i0n10          2
    3       3   yes      r212i0n10          3
    4       0   yes      r212i0n11          0
    5       1   yes      r212i0n11          1
    6       2   yes      r212i0n11          2
    7       3   yes      r212i0n11          3
```

The second problem is that, as demonstrated with the **ps** command below, the OpenMP threads are also placed on the same core where the associated MPI process is running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp


PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:02  4098  4092  4098
  0 a.out       00:00:02  4098  4092  4108
  0 a.out       00:00:02  4098  4092  4110
  1 a.out       00:00:03  4099  4092  4099
  1 a.out       00:00:03  4099  4092  4106
  2 a.out       00:00:03  4100  4092  4100
  2 a.out       00:00:03  4100  4092  4109
  3 a.out       00:00:03  4101  4092  4101
  3 a.out       00:00:03  4101  4092  4107
```

Sample PBS script demonstrating proper placement:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load mpi-hpe/mpt
module load comp-intel/2015.0.090

setenv MPI_DSM_VERBOSE
setenv OMP_NUM_THREADS 2
setenv KMP_AFFINITY disabled
```

```
cd $PBS_O_WORKDIR

#the following two lines will result in identical placement

mpiexec -np 8 omplace -nt 2 -c 0,1,3,4,6,7,9,10 -vv ./a.out
#mpiexec -np 8 omplace -nt 2 -c 0-10:bs=2+st=3 -vv  ./a.out
```

Shown in the PBS stderr file, the 4 MPI processes on each node are properly distributed on the two sockets with processes 0 and 1 on CPUs 0 and 3 (first socket) and processes 2 and 3 on CPUs 6 and 9 (second socket).

```
MPI: DSM information
MPI: using dplace
grank   lrank   pinning  node name      cpuid
    0       0   yes      r212i0n10          0
    1       1   yes      r212i0n10          3
    2       2   yes      r212i0n10          6
    3       3   yes      r212i0n10          9
    4       0   yes      r212i0n11          0
    5       1   yes      r212i0n11          3
    6       2   yes      r212i0n11          6
    7       3   yes      r212i0n11          9
```

In the PBS stout file, it shows the placement of the two OpenMP threads for each MPI process:

```
omplace: This is an HPE MPI program.
omplace: placement file /tmp/omplace.file.6454
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
omplace: This is an HPE MPI program.
omplace: placement file /tmp/omplace.file.22771
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
```

To get a better picture of how the OpenMP threads are placed, using the following **ps** command:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp

PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:06  4436  4435  4436
  1 a.out       00:00:03  4436  4435  4447
  1 a.out       00:00:03  4436  4435  4448
  3 a.out       00:00:06  4437  4435  4437
  4 a.out       00:00:05  4437  4435  4446
  6 a.out       00:00:06  4438  4435  4438
  7 a.out       00:00:05  4438  4435  4444
  9 a.out       00:00:06  4439  4435  4439
 10 a.out       00:00:05  4439  4435  4445
```

Process/Thread Pinning Overview

Pinning, the binding of a process or thread to a specific core, can improve the performance of your code by increasing the percentage of local memory accesses.

Once your code runs and produces correct results on a system, the next step is performance improvement. For a code that uses multiple cores, the placement of processes and/or threads can play a significant role in performance.

Given a set of processor cores in a PBS job, the Linux kernel usually does a reasonably good job of mapping processes/threads to physical cores, although the kernel may also migrate processes/threads. Some OpenMP runtime libraries and MPI libraries may also perform certain placements by default. In cases where the placements by the kernel or the MPI or OpenMP libraries are not optimal, you can try several methods to control the placement in order to improve performance of your code. Using the same placement from run to run also has the added benefit of reducing runtime variability.

Pay attention to maximizing data locality while minimizing latency and resource contention, and have a clear understanding of the characteristics of your own code and the machine that the code is running on.

Characteristics of NAS Systems

NAS provides two distinctly different types of systems: Pleiades, Aitken, and Electra are cluster systems, and Endeavour is a global shared-memory system. Each type is described in this section.

Pleiades, Aitken, and Electra

On Pleiades, Aitken, and Electra, memory on each node is accessible and shared only by the processes and threads running on that node. Pleiades is a cluster system consisting of different processor types: Sandy Bridge, Ivy Bridge, Haswell, and Broadwell. Electra is a cluster system that consists of Broadwell and Skylake nodes, and Aitken is a cluster system that consists of Cascade Lake and AMD Rome nodes.

Each node contains two sockets, with a symmetric memory system inside each socket. These nodes are considered non-uniform memory access (NUMA) systems, and memory is accessed across the two sockets through the inter-socket interconnect. So, for optimal performance, data locality should not be overlooked on these processor types.

However, compared to a global shared-memory NUMA system such as Endeavour, data locality is less of a concern on the cluster systems. Rather, minimizing latency and resource contention will be the main focus when pinning processes/threads on these systems.

For more information on Pleiades, Aitken, and Electra, see the following articles:

- Pleiades Configuration Details
- Aitken Configuration Details
- Electra Configuration Details

Endeavour

Endeavour comprises two hosts. Each host is a NUMA system that contains 32 sockets with a total of 896 cores. A process/thread can access the local memory on its socket, remote memory across sockets within the same chassis through the Ultra Path Interconnnect, and remote memory across chassis through the HPE Superdome Flex ASICs, with varying latencies. So, data locality is critical for achieving good performance on Endeavour.

Note: When developing an application, we recommend that you initialize data in parallel so that each processor core initializes the data it is likely to access later for calculation.

For more information, see Endeavour Configuration Details.

Methods for Process/Thread Pinning

Several pinning approaches for OpenMP, MPI and MPI+OpenMP hybrid applications are listed below. We recommend using the Intel compiler (and its runtime library) and the HPE MPT software on NAS systems, so most of the approaches pertain specifically to them. You can also use the `mbind` tool for multiple OpenMP libraries and MPI environments.

OpenMP codes

- Using Intel OpenMP Thread Affinity for Pinning
- Using the dplace Tool for Pinning
- Using the omplace Tool for Pinning
- Using the mbind Tool for Pinning

MPI codes

- Setting HPE MPT Environment Variables

- Using the omplace Tool for Pinniing
- Using the mbind Tool for Pinning

MPI+OpenMP hybrid codes

- Using the omplace Tool for Pinning
- Using the mbind Tool for Pinning

Checking Process/Thread Placement

Each of the approaches listed above provides some verbose capability to print out the tool's placement results. In addition, you can check the placement using the following approaches.

Use the ps Command

```
ps -C executable_name -L -opsr,comm,time,pid,ppid,lwp
```

In the generated output, use the core ID under the PSR column, the process ID under the PID column, and the thread ID under the LWP column to find where the processes and/or threads are placed on the cores.

Note: The **`ps`** command provides a snapshot of the placement at that specific time. You may need to monitor the placement from time to time to make sure that the processes/threads do not migrate.

Instrument your code to get placement information

- Call the **`mpi_get_processor_name`** function to get the name of the processor an MPI process is running on
- Call the Linux C function **`sched_getcpu()`** to get the processor number that the process or thread is running on

For more information, see Instrumenting your Fortran Code to Check Process/Thread Placement.

Hyperthreading

Hyperthreading is available and enabled on the Pleiades, Aitken, and Electra compute nodes. With hyperthreading, each physical core can function as two logical processors. This means that the operating system can assign two threads per core by assigning one thread to each logical processor.

Note: Hyperthreading is currently off on Aitken Rome nodes.

The following table shows the number of physical cores and potential logical processors available for each processor type:

Processor Model	Physical Cores (N)	Logical Processors (2N)
Sandy Bridge	16	32
Ivy Bridge	20	40
Haswell	24	48
Broadwell	28	56
Skylake	40	80
Cascade Lake	40	80

If you use hyperthreading, you can run an MPI code using 2*N* processes per node instead of *N* process per nodeâ   so you can use half the number of nodes for your job. Each process will be assigned to run on one logical processor; in reality, two processes are running on the same physical core.

Running two processes per core can take less than twice the wall-clock time compared to running only one process per coreâ   if one process does not keep the functional units in the core busy, and can share the resources in the core with another process.

Benefits and Drawbacks

Using hyperthreading can improve the overall throughput of your jobs, potentially saving standard billing unit (SBU) charges. Also, requesting half the usual number of nodes may allow your job to start running soonerâ   an added benefit when the systems are loaded with many jobs. However, using hyperthreading may not always result in better performance.

WARNING: Hyperthreading does not benefit all applications. Also, some applications may show improvement with some process counts but not with others, and there may be other unforeseen issues. Therefore, before using this technology in your production run, you should test your applications with and without hyperthreading. If your application runs more than two times slower with hyperthreading than without, do not use it.

Using Hyperthreading

Hyperthreading can improve the overall throughput, as demonstrated in the following example.

Example

Consider the following scenario with a job that uses 40 MPI ranks on Ivy Bridge. Without hyperthreading, we would specify:

```
#PBS -lselect=2:ncpus=20:mpiprocs=20:model=ivy
```

and the job will use 2 nodes with 20 processes per node. Suppose that the job takes 1000 seconds when run this way. If we run the job with hyperthreading, e.g.:

```
#PBS -lselect=1:ncpus=20:mpiprocs=40:model=ivy
```

then the job will use 1 node with all 40 processes running on that node. Suppose this job takes 1800 seconds to complete.

Without hyperthreading, we used 2 nodes for 1000 seconds (a total of 2000 node-seconds); with hyperthreading, we used 1 node for 1800 seconds (1800 node-seconds). Thus, under these circumstances, if you were interested in getting the best wall-clock time performance for a single job, you would use two nodes without hyperthreading. However, if you were interested in minimizing resource usage, especially with multiple jobs running simultaneously, using hyperthreading would save you 10% in SBU charges.

Mapping of Physical Core IDs and Logical Processor IDs

Mapping between the physical core IDs and the logical processor IDs is summarized in the following table. The value of *N* is 16, 20, 24, 28, and 40 for Sandy Bridge, Ivy Bridge, Haswell, Broadwell, and Skylake/Cascade Lake processor types, respectively.

Physical ID	Physical Core ID	Logical Processor ID
0	0	0 ; *N*
0	1	1 ; *N*+1
...	...	....
0	*N*/2 - 1	*N*/2 - 1; *N* + *N*/2 - 1
1	*N*/2	*N*/2 ; *N* + *N*/2
1	*N*/2 + 1	*N*/2 + 1; *N* + *N*/2 + 1
1	...	...
1	*N* - 1	*N* - 1; 2*N* - 1

Note: For additional mapping details, see the configuration diagrams at the for each processor type, or run the **`cat /proc/cpuinfo`** command on the specific node type.

Using the dplace Tool for Pinning

The **`dplace`** tool binds processes/threads to specific processor cores to improve your code performance. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

Once pinned, the processes/threads do not migrate. This can improve the performance of your code by increasing the percentage of local memory accesses.

`dplace` invokes a kernel module to create a job placement container consisting of all (or a subset of) the CPUs of the cpuset. In the current **`dplace`** version 2, an LD_PRELOAD library (libdplace.so) is used to intercept calls to the functions **`fork()`**, **`exec(`**), and **`pthread_create()`** to place tasks that are being created. Note that tasks created internal to glib are not intercepted by the preload library. These tasks will *not* be placed. If no placement file is being used, then the **`dplace`** process is placed in the job placement container and (by default) is bound to the first CPU of the cpuset associated with the container.

Syntax

```
dplace [-e] [-c cpu_numbers] [-s skip_count] [-n process_name] \
           [-x skip_mask] [-r [l|b|t]] [-o log_file] [-v 1|2] \
           command [command-args]
dplace [-p placement_file] [-o log_file] command [mpiexec -np4 a.out]
dplace [-q] [-qq] [-qqq]
```

As illustrated above, **`dplace`** "execs" command (in this case, without its **`mpiexec`** arguments), which executes within this placement container and continues to be bound to the first CPU of the container. As the command forks child processes, they inherit the container and are bound to the next available CPU of the container.

If a placement file is being used, then the **`dplace`** process is not placed at the time the job placement container is created. Instead, placement occurs as processes are forked and executed.

Options for dplace

Explanations for some of the options are provided below. For additional information, see **man dplace**.

-e and -c *cpu_numbers*

`-e` determines exact placement. As processes are created, they are bound to CPUs in the exact order specified in the CPU list. CPU numbers may appear multiple times in the list.

A CPU value of "x" indicates that binding should *not* be done for that process. If the end of the list is reached, binding starts over again at the beginning of the list.

`-c` ***`cpu_numbers`*** specifies a list of CPUs, optionally strided CPU ranges, or a striding pattern. For example:

- -c 1
- -c 2-4 (equivalent to -c 2,3,4)
- -c 12-8 (equivalent to -c 12,11,10,9,8)
- -c 1,4-8,3

- -c 2-8:3 (equivalent to -c 2,5,8)
- -c CS
- -c BT

Note: CPU numbers are *not* physical CPU numbers. They are logical CPU numbers that are relative to the CPUs that are in the allowed set, as specified by the current cpuset.

A CPU value of "x" (or *), in the argument list for the **`-c`** option, indicates that binding should not be done for that process. The value "**`x`**" should be used only if the **`-e`** option is also used.

Note that CPU numbers start at 0.

For striding patterns, any subset of the characters (**`B`**)lade, (**`S`**)ocket, (**`C`**)ore, (**`T`**)hread may be used; their ordering specifies the nesting of the iteration. For example, **`SC`** means to iterate all the cores in a socket before moving to the next CPU socket, while **`CB`** means to pin to the first core of each blade, then the second core of every blade, and so on.

For best results, use the **`-e`** option when using stride patterns. If the **`-c`** option is not specified, all CPUs of the current cpuset are available. The command itself (which is "execed" by **`dplace`**) is the first process to be placed by the **`-c`** ***`cpu_numbers`***.

Without the **`-e`** option, the order of numbers for the **`-c`** option is not important.

`-x` ***`skip_mask`***
: Provides the ability to skip placement of processes. The ***`skip_mask`*** argument is a bitmask. If bit *N* of ***`skip_mask`*** is set, then the *N*+1th process that is forked is not placed. For example, setting the mask to 6 prevents the second and third processes from being placed. The first process (the process named by the command) will be assigned to the first CPU. The second and third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time.

`-s` ***`skip_count`***
: Skips the first ***`skip_count`*** processes before starting to place processes onto CPUs. This option is useful if the first ***`skip_count`*** processes are "shepherd" processes used only for launching the application. If ***`skip_count`*** is not specified, a default value of 0 is used.

`-q`
: Lists the global count of the number of active processes that have been placed (by **`dplace`**) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, not physical CPU numbers. If specified twice, lists the current **`dplace`** jobs that are running. If specified three times, lists the current **`dplace`** jobs and the tasks that are in each job.

`-o` ***`log_file`***
: Writes a trace file to ***`log_file`*** that describes the placement actions that were made for each fork, exec, etc. Each line contains a time-stamp, process id:thread number, CPU that task was executing on, taskname and placement action. Works with version 2 only.

Examples of dplace Usage

For OpenMP Codes

```
#PBS -lselect=1:ncpus=8

#With Intel compiler versions 10.1.015 and later,
#you need to set KMP_AFFINITY to disabled
#to avoid the interference between dplace and
#Intel's thread affinity interface.
```

```
setenv KMP_AFFINITY disabled

#The -x2 option provides a skip map of 010 (binary 2) to
#specify that the 2nd thread should not be bound. This is
#because under the new kernels, the master thread (first thread)
#will fork off one monitor thread (2nd thread) which does
#not need to be pinned.

#On Pleiades, if the number of threads is less than
#the number of cores, choose how you want
#to place the threads carefully. For example,
#the following placement is good on Harpertown
#but not good on other Pleiades processor types:

dplace -x2 -c 2,1,4,5 ./a.out
```

To check the thread placement, you can add the **`-o`** option to create a log:

```
dplace -x2 -c 2,1,4,5 -o log_file ./a.out
```

Or use the following command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample Output of log_file

```
timestamp       process:thread cpu taskname| placement action
15:32:42.196786 31044          1 dplace    | exec ./openmp1, ncpu 1
15:32:42.210628 31044:0        1 a.out     | load, cpu 1
15:32:42.211785 31044:0        1 a.out     | pthread_create thread_number 1, ncpu -1
15:32:42.211850 31044:1        - a.out     | new_thread
15:32:42.212223 31044:0        1 a.out     | pthread_create thread_number 2, ncpu 2
15:32:42.212298 31044:2        2 a.out     | new_thread
15:32:42.212630 31044:0        1 a.out     | pthread_create thread_number 3, ncpu 4
15:32:42.212717 31044:3        4 a.out     | new_thread
15:32:42.213082 31044:0        1 a.out     | pthread_create thread_number 4, ncpu 5
15:32:42.213167 31044:4        5 a.out     | new_thread
15:32:54.709509 31044:0        1 a.out     | exit
```

Sample Output of placement.out

```
PSR COMMAND             TIME   PID  PPID   LWP
  1 a.out           00:00:02 31044 31039 31044
  0 a.out           00:00:00 31044 31039 31046
  2 a.out           00:00:02 31044 31039 31047
  4 a.out           00:00:01 31044 31039 31048
  5 a.out           00:00:01 31044 31039 31049
```

Note: Intel OpenMP jobs use an extra thread that is unknown to the user and it does not need to be placed. In the above example, this extra thread (31046) is running on core number 0.

For MPI Codes Built with HPE's MPT Library

With HPE's MPT, only 1 shepherd process is created for the entire pool of MPI processes, and the proper way of pinning using **`dplace`** is to skip the shepherd process.

Here is an example for Endeavour:

```
#PBS -l ncpus=8
....
```

```
 mpiexec -np 8 dplace -s1 -c 0-7 ./a.out
```

On Pleiades, if the number of processes in each node is less than the number of cores in that node, choose how you want to place the processes carefully. For example, the following placement works well on Harpertown nodes, but not on other Pleiades processor types:

```
#PBS -l select=2:ncpus=8:mpiprocs=4 ... mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

To check the placement, you can set MPI_DSM_VERBOSE, which prints the placement in the PBS stderr file:

```
#PBS -l select=2:ncpus=8:mpiprocs=4
...
setenv MPI_DSM_VERBOSE
mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

Output in PBS stderr File

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
    0       0   yes      r75i2n13          1
    1       1   yes      r75i2n13          2
    2       2   yes      r75i2n13          4
    3       3   yes      r75i2n13          5
    4       0   yes      r87i2n6           1
    5       1   yes      r87i2n6           2
    6       2   yes      r87i2n6           4
    7       3   yes      r87i2n6           5
```

If you use the `-o` ***`log_file`*** flag of **`dplace`**, the CPUs where the processes/threads are placed will be printed, but the node names are not printed.

```
#PBS -l select=2:ncpus=8:mpiprocs=4
....
mpiexec -np 8 dplace -s1 -c 2,4,1,5 -o log_file ./a.out
```

Output in log_file

```
timestamp       process:thread cpu taskname | placement action
15:16:35.848646 19807          - dplace     | exec ./new_pi, ncpu -1
15:16:35.877584 19807:0        - a.out      | load, cpu -1
15:16:35.878256 19807:0        - a.out      | fork -> pid 19810, ncpu 1
15:16:35.879496 19807:0        - a.out      | fork -> pid 19811, ncpu 2
15:16:35.880053 22665:0        - a.out      | fork -> pid 22672, ncpu 2
15:16:35.880628 19807:0        - a.out      | fork -> pid 19812, ncpu 4
15:16:35.881283 22665:0        - a.out      | fork -> pid 22673, ncpu 4
15:16:35.882536 22665:0        - a.out      | fork -> pid 22674, ncpu 5
15:16:35.881960 19807:0        - a.out      | fork -> pid 19813, ncpu 5
15:16:57.258113 19810:0        1 a.out      | exit
15:16:57.258116 19813:0        5 a.out      | exit
15:16:57.258215 19811:0        2 a.out      | exit
15:16:57.258272 19812:0        4 a.out      | exit
15:16:57.260458 22672:0        2 a.out      | exit
15:16:57.260601 22673:0        4 a.out      | exit
15:16:57.260680 22674:0        5 a.out      | exit
15:16:57.260675 22671:0        1 a.out      | exit
```

For MPI Codes Built with MVAPICH2 Library

With MVAPICH2, 1 shepherd process is created for each MPI process. You can use **`ps -L -u`** ***`your_userid`*** on the running node to see these processes. To properly pin MPI processes using **`dplace`**, you cannot skip the shepherd processes and must use the following:

```
mpiexec -np 4 dplace -c2,4,1,5 ./a.out
```

Using the mbind Tool for Pinning

The **`mbind`** utility is a "one-stop" tool for binding processes and threads to CPUs. It can also be used to track memory usage. The utility, developed at NAS, works for for MPI, OpenMP, and hybrid applications, and is available in the **`/u/scicon/tools/bin`** directory on Pleiades.

Recommendation: Add **`/u/scicon/tools/bin`** to the PATH environment variable in your startup configuration file to avoid having to include the entire path in the command line.

One of the benefits of **`mbind`** is that it relieves you from having to learn the complexity of each individual pinning approach for associated MPI or OpenMP libraries. It provides a uniform usage model that works for multiple MPI and OpenMP environments.

Currently supported MPI and OpenMP libraries are listed below.

MPI:

- HPE-MPT
- MVAPICH2
- INTEL-MPI
- OPEN-MPI (including Mellanox HPC-X MPI)
- MPICH

Note: When using **`mbind`** with HPE-MPT, it is highly recommended that you use MPT 2.17r13, 2.21 or a later version in order to take full advantage of **`mbind`** capabilities.

OpenMP:

- Intel OpenMP runtime library
- GNU OpenMP library
- PGI runtime library
- Oracle Developer Studio thread library

Starting with version 1.7, the use of **`mbind`** is no longer limited to cases where the same set of CPU lists is used for all processor nodes. However, as in previous versions, the same number of threads must be used for all processes.

WARNING: The **`mbind`** tool might not work properly when used together with other performance tools.

Syntax

```
#For OpenMP:
mbind.x [-options] program [args]

#For MPI or MPI+OpenMP hybrid which supports mpiexec:
mpiexec -np nranks mbind.x [-options] program [args]
```

To find information about all available options, run the command **`mbind.x -help`**.

Here are a few recommended **`mbind`** options:

`-cs, -cp, -cc;`
`or -c`*`cpulist`*

  `-cs` for spread (default), **`-cp`** for compact, **`-cc`** for cyclic; **`-ccpulist`** for process ranks (for example, **`-c`*`0,3,6,9`***). CPU numbers in the *cpulist* are relative within a **`cpuset`**, if present.

Note that the **-cs** option will distribute the processes and threads among the physical cores to minimize various resource contentions, and is usually the best choice for placement.

-t[*n*]
: Number of threads per process. The default value is given by the OMP_NUM_THREADS environment variable; this option overrides the value specified by OMP_NUM_THREADS.

-gm[*n*]
: Print memory usage information. This option is for printing memory usage of each process at the end of a run. Optional value [*n*] can be used to select one of the memory usage types: **0=default, 1=VmHWM, 2=VmRSS, 3=WRSS**. Recognized symbolic values for [*n*]: "**hwm**", "**rss**", or "**wrss**". For **default**, environment variable GM_TYPE may be used to select the memory usage type:

  ◊ **VmHWM** - high water mark
  ◊ **VmRSS** - resident memory size
  ◊ **WRSS** - weighted memory usage (if available; else, same as **VmRSS**)

-gmc[*s:n*]
: Print memory usage every [*s*] seconds for [*n*] times. The **-gmc** option indicates continuous printing of memory usage at a default interval of 5 seconds. Use additional option [*s:n*] to control the interval length [*s*] and the number of printing times [*n*]. Environment variable GM_TIMER may also be used to set the [*s:n*] value.

-gmr[*list*]
: Print memory usage for selected ranks in the list. This option controls the subset of ranks for memory usage to print. [*list*] is a comma-separated group of numbers with possible range.

-l
: Print node information. This option prints a quick summary of node information by calling the **clist.x** utility.

-v[*n*]
: Verbose flag; Option **-v** or **-v1** prints the process/thread-CPU binding information. With **[*n*]** greater than 1, the option prints additional debugging information. **[*n*]** controls the level of details. Default is **n=0** (OFF).

Examples

Print a Processor Node Summary to Help Determine Proper Process or Thread Pinning

In your PBS script, add the following to print the summary:

```
#PBS -lselect=...:model=bro

mbind.x -l

...
```

In the sample output below for a Broadwell node, look for the listing under column **CPUs(SMTs)**. CPUs listed in the same row are located in the same socket and share the same last level cache, as shown in this configuration diagram.

```
Host Name           : r601i0n3
Processor Model     : Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
Processor Speed     : 1600 MHz (max 2401 MHz)
Level 1 Cache (D)   : 32 KB
Level 1 Cache (I)   : 32 KB
Level 2 Cache (U)   : 256 KB
Level 3 Cache (U)   : 35840 KB (shared by 14 cores)
SMP Node Memory     : 125.0 GB (122.3 GB free, 2 mem nodes)

Number of Sockets   : 2
Number of L3 Caches : 2
Number of Cores     : 28
Number of SMTs/Core : 2
Number of CPUs      : 56

Socket  Cache   Cores           CPUs(SMTs)
0       0       0-6,8-14        (0,28)(1,29)(2,30)(3,31)(4,32)(5,33)(6,34)(7,35)(8,36)
                                (9,37)(10,38)(11,39)(12,40)(13,41)
1       0       0-6,8-14        (14,42)(15,43)(16,44)(17,45)(18,46)(19,47)(20,48)(21,49)
                                (22,50)(23,51)(24,52)(25,53)(26,54)(27,55)
```

For Pure OpenMP Codes Using Intel OpenMP Library

Sample PBS script:

```
#PBS -l select=1:ncpus=28:model=bro
#PBS -l walltime=0:5:0

module load comp-intel

setenv OMP_NUM_THREADS 4
cd $PBS_O_WORKDIR

mbind.x -cs -t4 -v ./a.out
#or simply:
#mbind.x -v ./a.out
```

The four OpenMP threads are spread (with the **`-cs`** option) among four physical cores in a node (two on each socket), as shown in the application's stdout:

```
host: r635i7n14, ncpus: 56, nthreads: 4, bound to cpus: {0,1,14,15}
OMP: Warning #181: GOMP_CPU_AFFINITY: ignored because KMP_AFFINITY has been defined
```

The proper placement is further demonstrated in the output of the **`ps`** command below:

```
r635i7n14% ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND             TIME   PID  PPID   LWP
  0 a.out           00:02:06 37243 36771 37243
  1 a.out           00:02:34 37243 36771 37244
 14 a.out           00:01:47 37243 36771 37245
 15 a.out           00:01:23 37243 36771 37246
```

Note: If you use older versions of Intel OpenMP via older versions of Intel compiler modules (comp-intel/2016.181 or earlier) during runtime, the **`ps`** output will show an extra thread that does not do any work, and therefore does not accumulate any time. Since this extra thread will not interfere with the other threads, it does not need to be placed.

For Pure MPI Codes Using HPE MPT

WARNING: **`mbind.x`** disables MPI_DSM_DISTRIBUTE and overwrites the placement initially performed by MPT's **`mpiexec`**. The placement output from MPI_DSM_VERBOSE (if set) most likely is incorrect and should be ignored.
Sample PBS script where the same number of MPI ranks are used in different nodes:

```
#PBS -l select=2:ncpus=28:mpiprocs=4:model=bro

module load comp-intel
module load mpi-hpe

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -v ./a.out
#or simply:
```

```
#mpiexec mbind.x -v ./a.out
```

On each of the two nodes, four MPI processes are spread among four physical cores (CPUs 0,1,14,15); two on each socket, as shown in the application's stdout:

```
host: r601i0n3, ncpus: 56, process-rank: 0 (r0), bound to cpu: 0
host: r601i0n3, ncpus: 56, process-rank: 1 (r1), bound to cpu: 1
host: r601i0n3, ncpus: 56, process-rank: 2 (r2), bound to cpu: 14
host: r601i0n3, ncpus: 56, process-rank: 3 (r3), bound to cpu: 15
host: r601i0n4, ncpus: 56, process-rank: 4 (r0), bound to cpu: 0
host: r601i0n4, ncpus: 56, process-rank: 5 (r1), bound to cpu: 1
host: r601i0n4, ncpus: 56, process-rank: 6 (r2), bound to cpu: 14
host: r601i0n4, ncpus: 56, process-rank: 7 (r3), bound to cpu: 15
```

Note: For readability in this article, the printout of the binding information from **`mbind.x`** is sorted by the process-rank. An actual printout will not be sorted.

Sample PBS script where different numbers of MPI ranks are used on different nodes:

```
#PBS -l select=1:ncpus=28:mpiprocs=1:model=bro+2:ncpus=28:mpiprocs=4:model=bro

module load comp-intel
module load mpi-hpe

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 9 mbind.x -cs -v ./a.out
#Or simply:
#mpiexec mbind.x -v ./a.out
```

As shown in the application's stdout, only one MPI process is used on the first node and it is pinned to CPU 0 on that node. For each of the other two nodes, four MPI processes are spread among four physical cores (CPUs 0,1,14,15):

```
host: r601i0n3, ncpus: 56, process-rank: 0 (r0), bound to cpu: 0
host: r601i0n4, ncpus: 56, process-rank: 1 (r0), bound to cpu: 0
host: r601i0n4, ncpus: 56, process-rank: 2 (r1), bound to cpu: 1
host: r601i0n4, ncpus: 56, process-rank: 3 (r2), bound to cpu: 14
host: r601i0n4, ncpus: 56, process-rank: 4 (r3), bound to cpu: 15
host: r601i0n12, ncpus: 56, process-rank: 5 (r0), bound to cpu: 0
host: r601i0n12, ncpus: 56, process-rank: 6 (r1), bound to cpu: 1
host: r601i0n12, ncpus: 56, process-rank: 7 (r2), bound to cpu: 14
host: r601i0n12, ncpus: 56, process-rank: 8 (r3), bound to cpu: 15
```

For MPI+OpenMP Hybrid Codes Using HPE MPT and Intel OpenMP

Sample PBS script:

```
#PBS -l select=2:ncpus=28:mpiprocs=4:model=bro

module load comp-intel
module load mpi-hpe

setenv OMP_NUM_THREADS 2
#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -t2 -v ./a.out
#or simply:
#mpiexec mbind.x -v ./a.out
```

On each of the two nodes, the four MPI processes are spread among the physical cores. The two OpenMP threads of each MPI process run on adjacent physical cores, as shown in the application's stdout:

```
host: r623i5n2, ncpus: 56, process-rank: 0 (r0), nthreads: 2, bound to cpus: {0,1}
host: r623i5n2, ncpus: 56, process-rank: 1 (r1), nthreads: 2, bound to cpus: {2,3}
host: r623i5n2, ncpus: 56, process-rank: 2 (r2), nthreads: 2, bound to cpus: {14,15}
host: r623i5n2, ncpus: 56, process-rank: 3 (r3), nthreads: 2, bound to cpus: {16,17}
host: r623i6n9, ncpus: 56, process-rank: 4 (r0), nthreads: 2, bound to cpus: {0,1}
host: r623i6n9, ncpus: 56, process-rank: 5 (r1), nthreads: 2, bound to cpus: {2,3}
host: r623i6n9, ncpus: 56, process-rank: 6 (r2), nthreads: 2, bound to cpus: {14,15}
host: r623i6n9, ncpus: 56, process-rank: 7 (r3), nthreads: 2, bound to cpus: {16,17}
```

You can confirm this by running the following **ps** command line on the running nodes. Note that the HPE MPT library creates a shepherd process (shown running on **PSR=18** in the output below), which does not do any work.

```
r623i5n2% ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND     TIME   PID  PPID   LWP
 18 a.out   00:00:00 41087 41079 41087
  0 a.out   00:00:12 41092 41087 41092
  1 a.out   00:00:12 41092 41087 41099
  2 a.out   00:00:12 41093 41087 41093
  3 a.out   00:00:12 41093 41087 41098
 14 a.out   00:00:12 41094 41087 41094
 15 a.out   00:00:12 41094 41087 41097
 16 a.out   00:00:12 41095 41087 41095
 17 a.out   00:00:12 41095 41087 41096
```

For Pure MPI or MPI+OpenMP Hybrid Codes Using other MPI Libraries and Intel OpenMP

Usage of **mbind** with MPI libraries such as HPC-X or Intel-MPI should be the same as with HPE MPT. The main difference is that you must load the proper **mpi** modulefile, as follows:

- For HPC-X:

  ```
  module load mpi-hpcx
  ```
- For Intel-MPI:

  ```
  module use /nasa/modulefiles/testing
  module load mpi-intel
  ```

  Note that the Intel MPI library automatically pins processes to CPUs to prevent unwanted process migration. If you find that the placement done by the Intel MPI library is not optimal, you can use **mbind** to do the pinning instead. If you use version 4.0.2.003 or earlier, you might need to set the environment variable I_MPI_PIN to **0** in order for **mbind.x** to work properly.

*The **mbind** utility was created by NAS staff member Henry Jin.*